Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Sistema de Arquivos Distribuído Tolerante a Falhas Maliciosas: Uma Prova de Conceito

Guilherme Teixeira Soares
Pedro Henrique Vidal Pinho

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Me. Pedro Antônio Dourado Rezende

Banca examinadora composta por:

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador) — CIC/UnB
Prof. Dr. André Costa Drummond — CIC/UnB
Prof. Me. João José Costa Gondim — CIC/UnB

CIP — Catalogação Internacional na Publicação

Soares, Guilherme Teixeira.

Sistema de Arquivos Distribuído Tolerante a Falhas Maliciosas: Uma Prova de Conceito / Guilherme Teixeira Soares, Pedro Henrique Vidal Pinho. Brasília : UnB, 2016.
137 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2016.

1. sistemas distribuídos, 2. arquivo, 3. segurança, 4. programação

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Sistema de Arquivos Distribuído Tolerante a Falhas Maliciosas: Uma Prova de Conceito

Guilherme Teixeira Soares
Pedro Henrique Vidal Pinho

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador)
CIC/UnB

Prof. Dr. André Costa Drummond
CIC/UnB

Prof. Me. João José Costa Gondim
CIC/UnB

Prof. Me. Pedro Antônio Dourado Rezende
Coordenador do Curso de Computação — Licenciatura

Brasília, 04 de julho de 2016

Dedicatória

Dedicamos este trabalho a todas as pessoas interessadas no assunto de Sistema de Arquivos Distribuído.

Agradecimentos

Agradecemos primeiramente a Deus, pois sem ele jamais teríamos conseguido chegar ao fim do nosso curso. As nossas famílias, que sempre nos apoiaram e ajudaram quando precisamos. Agradecemos ao professor Eduardo Alchieri, pelos ensinamentos durante toda a graduação. Por último, agradecemos a todas as amizades feitas ao longo do curso.

Resumo

Sistema de Arquivos Distribuído possuí como principal característica, propor ao usuário a ilusão de que as requisições feitas ao sistema, estão sendo tratadas em um único local. Contudo, nesse tipo de sistema, vários são os componentes que compõem o sistema como um todo. Um mesmo dado é armazenado em diferentes locais e de diferentes formas. Lidar com esse tipo de sistema, nos proporciona uma certa atenção, no que diz respeito a segurança. Uma vez que, os dados dos arquivos estão armazenados em diferentes locais, maior será a atenção em relação a segurança e gerenciamento do sistema. Propomos a criação de um Sistema de Arquivos Distribuído baseado na biblioteca BFT-SMaRt, que seja tolerante a falhas maliciosas. Uma contribuição do nosso sistema visa abordar as diferentes metodologias para a escrita e leitura de arquivos, nesse tipo de sistema.

Palavras-chave: sistemas distribuídos, arquivo, segurança, programação

Abstract

Distributed File System have as their main characteristic, offer the user the illusion that the requests made to the system, are being processed in a single location. However, in this type of system, there are many components that make up the system. The same data is stored in different locations and in different ways. Handling this type of system gives us a certain attention, regarding security. Once the files are stored in different locations, greater will be the attention given to security and system management. We propose the creation of a Distributed File System based on the BFT-SMaRt library, which is tolerant of malicious faults. A contribution by our system is to aim to address the different methods for writing and reading files in this kind of system.

Keywords: distributed systems, file, security, programming

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

Capítulo 1

Introdução

Na computação, a maioria dos aplicativos que estão em execução, precisam armazenar e recuperar informações. Os processos desses aplicativos podem utilizar os próprios espaços de endereçamento para armazenar dados, mas a utilização desses espaços para armazenamento possuem algumas limitações e problemas:

- O espaço de memória reservado para os processos são pequenos, o que impede o armazenamento de uma grande quantidade de dados, que hoje é a necessidade da maioria dos programas;
- Ao processo ser finalizado ou o computador desligado as informações são perdidas;
- Apenas o próprio processo pode acessar as informações no seu espaço de endereçamento, o que é inviável para processos que precisam frequentemente acessar os mesmos dados.

Esses problemas e limitações são resolvidos armazenando informações em memórias não voláteis, como discos e outras mídias externas, em unidades chamadas arquivos. Essas informações armazenadas em arquivos devem ser persistentes, isto é, não devem ser afetadas pela criação ou finalização de processos. O gerenciamento desses arquivos é realizado pelo sistema operacional (SO), mais especificamente pela parte do SO denominada Sistema de Arquivos. O modo em que os arquivos são estruturados, nomeados, acessados, utilizados, protegidos e implementados é função do sistema de arquivos.

Do ponto de vista dos usuários, o aspecto mais importante do sistema de arquivos é a forma em que os arquivos são apresentados, isto é, o que constitui um arquivo, como os arquivos são nomeados e protegidos, que operações são permitidas nesses arquivos e assim por diante.

Os sistemas de arquivos mais comuns são aqueles em que os dados são armazenados localmente, no próprio computador do usuário, e o acesso a esses dados está restrito apenas aos usuários que tem acesso a esse computador. Contudo, esses sistemas podem ser desenvolvidos de maneiras diferentes e para finalidades específicas, de modo que em cada situação, a forma que foi estruturado e implementado pode melhorar seu desempenho. O tipo de sistema de arquivos que será tema deste trabalho são os Sistemas de Arquivos Distribuídos que tem o objetivo de oferecer os mesmos serviços e recursos dos sistemas de arquivos convencionais, porém são estruturados de maneira diferente, principalmente

pela forma de armazenamento dos arquivos, pois estes são armazenados em *hardwares* diferentes interconectados através de uma rede.

Sistemas de Arquivos Distribuídos permitem que programas armazenem e acessem arquivos remotos da mesma forma que fazem com arquivos locais, permitindo que usuários acessem arquivos de qualquer computador da rede. Esses sistemas devem ser transparentes, ou seja, o fato de os arquivos serem armazenados em *hardwares* diferentes e a forma com que esses arquivos são armazenados e acessados remotamente, não deve ser notada pelos usuários, isto é, eles devem ter a mesma impressão de estarem utilizando um sistema de arquivos local.

A principal motivação para criação de um sistema distribuído é o compartilhamento de recursos (no caso desse trabalho, os recursos são os arquivos). Além disso, esses sistemas geralmente são escaláveis, ou seja, possibilitam adicionar, de modo simples, novos componentes ao sistema. A redundância também é um ponto importante, pois a possibilidade de uma máquina da rede ficar indisponível por qualquer motivo, não inviabiliza o funcionamento do sistema, pois outra máquina assume seu papel. Outros benefícios do uso de sistemas de arquivos distribuídos serão citados nos próximos capítulos.

O modo que os sistemas de arquivos distribuídos são estruturados, implica em uma complexidade maior de desenvolvimento em relação aos sistemas de arquivos convencionais. Manter a consistência de determinado arquivo se torna mais difícil, pois os arquivos devem ser replicados em mais de uma máquina caso alguma venha a falhar. Manter a integridade e confidencialidade de um arquivo nesse tipo de sistema também é um desafio, visto que os arquivos estão disponíveis para acesso e atualizações constantes e simultâneas por vários usuários da rede e mesmo assim esses arquivos precisam estar iguais em suas réplicas. Além disso, devido a essa estrutura diferenciada, vários outros problemas de segurança e desempenho, que serão tratados nos próximos capítulos, afetam com maior rigor o projeto e desenvolvimento de um sistema de arquivos distribuído.

Projetar e manter um sistema distribuído seguro é uma tarefa difícil, diante da quantidade de ameaças que o mesmo pode encontrar. Manter seguro os dados que são processados e armazenados nesse tipo de sistema é um dos maiores desafios. Geralmente esse tipo de sistema é formado por muitas redes e componentes, o que dificulta manter o sistema seguro em todos os pontos. Muitos recursos de informação que se tornam disponíveis e são mantidos em sistemas distribuídos, têm um alto valor intrínseco para seus usuários. Portanto, sua segurança é de considerável importância.

Basicamente, para que um sistema distribuído possa ser considerado seguro, o mesmo deve contemplar 3 atributos: disponibilidade, integridade e confidencialidade. O projeto deste trabalho, busca a implementação de um sistema de arquivos distribuído e seguro, porém iremos implementar, primeiramente, apenas mecanismos que assegurem a integridade e disponibilidade do nosso sistema e como segundo passo, conforme será abordado nos trabalhos futuros, buscaremos a implementação de mecanismos que garantam a confidencialidade no sistema.

1.1 Objetivos

Nesta seção serão descritos os objetivos gerais e os objetivos específicos do trabalho.

1.1.1 Objetivo Geral

Desenvolver um Sistema de Arquivos Distribuído, utilizando a biblioteca BFT-SMaRt (responsável pela replicação dos dados e pelo protocolo que tolera falhas bizantinas). Os metadados e os dados físicos dos arquivos serão armazenados em máquinas diferentes e as operações sobre os arquivos serão gerenciadas pelo grupo de máquinas que possui os metadados replicados. O sistema deve ser escalável, tolerante a falhas maliciosas e eficiente.

1.1.2 Objetivos Específicos

- Estudar os conceitos de segurança e funcionamento de um Sistema de Arquivos Distribuído;
- Estudar a biblioteca BFT-SMaRt, utilizada para o desenvolvimento do sistema;
- Desenvolver a hierarquia lógica de diretórios do sistema, com base na biblioteca BFT-SMaRt;
- Desenvolver as operações do sistema referente a manipulação dos arquivos e diretórios. Exemplos: escrever, ler, excluir, listar, entre outros;
- Analisar a solução proposta através da execução de uma série de experimentos;

1.2 Organização do Texto

O restante desta monografia está organizado da seguinte forma:

1. O capítulo 2 apresenta uma visão geral sobre os Sistemas Distribuídos, além de descrever os desafios enfrentados no projeto e desenvolvimento desses sistemas.
2. O capítulo 3 trata sobre os Sistemas de Arquivos Distribuídos, apresentando os conceitos básicos e requisitos para o funcionamento adequado desse tipo de sistema. Aborda também os conceitos, requisitos e técnicas para garantir a tolerância a falhas e segurança nesses sistemas.
3. O capítulo 4 apresenta a biblioteca BFT-SMaRt, que implementa a replicação de máquinas de estados e tolerância a falhas bizantinas, além de descrever o seu funcionamento.
4. O capítulo 5 descreve a nossa proposta para criação de um sistema de arquivos distribuído seguro utilizando a biblioteca BFT-SMaRt.
5. O capítulo 6 conclui o trabalho. Além da conclusão da monografia também é apresentado os trabalhos futuros para melhorias e continuidade do projeto.

Capítulo 2

Sistemas Distribuídos

Este capítulo apresenta uma visão geral da natureza dos Sistemas Distribuídos(SDs), além de descrever sobre os desafios enfrentados, para garantir que o sistema funcione de forma correta.

Os Sistemas Distribuídos estão sendo utilizados cada vez mais e estão em constante processo de desenvolvimento e inovação. A viabilidade e facilidade da construção de sistemas de computação, compostos por grandes quantidades de computadores conectados através de redes de alta velocidade e a necessidade de compartilhamento de recursos (arquivos, impressoras, programas etc) impulsionam o uso e desenvolvimento desses sistemas. Um grande exemplo de sistema distribuído que é amplamente utilizado é a Internet, porém existem vários outros: redes de telefones móveis, redes corporativas, redes em campus, entre outros.

Existem várias definições de sistemas distribuídos, nenhuma delas é completa e geralmente divergem, mas cada uma expõe características importantes que compõem esses sistemas.

> "Um sistema distribuído é uma coleção de computadores autônomos que aparentam para os seus usuários como se fossem um único e coerente sistema."
>
> Andrew S. Tanenbaum

Dessa definição, dois aspectos importantes são percebidos:

- Componentes autônomos compõem os sistemas distribuídos;
- Os usuários, sejam pessoas ou programas, acham que estão usando um sistema único. Isso significa que os componentes autônomos precisam cooperar entre si.

Uma definição alternativa a de Tanenbaum, que também expõe características importantes desse tipo de sistemas é:

> "Um sistema no qual componentes de *hardware* ou de *software* localizados em uma rede de computadores se comunicam e coordenam suas ações apenas por passagem de mensagens."
>
> Coulouris

Dessa definição, levantamos as seguintes características:

- Concorrência: A capacidade do sistema de manipular recursos compartilhados concorrentemente, como arquivos, página Web, impressoras, entre outros.
- Falta de um relógio global: Os programas cooperam entre si, coordenando suas ações através de trocas de mensagens. A coordenação frequentemente depende de uma noção compartilhada do tempo em que as ações dos programas ocorrem. Entretanto, existem limites para a previsão com a qual os computadores podem sincronizar os seus relógios em uma rede. Não existe uma noção global única do tempo;
- Falhas de componentes independentes: A falha de um componente que compõe o sistema, um computador por exemplo, deve isolá-lo do sistema, mas não deve resultar na falha do sistema.

2.1 Desafios

Das definições e características citadas, surgem vários problemas para o projeto de sistemas distribuídos. Esses problemas implicam em desafios que devem ser enfrentados e superados, para que o sistema seja bem-sucedido. Nesta seção apresentamos os principais desafios para construção dos sistemas distribuídos [26] [13].

2.1.1 Heterogeneidade

Das duas definições citadas, não há nenhuma premissa em relação aos tipos de computadores que compõem um sistema distribuído, i.e., em um mesmo sistema pode existir computadores com *hardwares* totalmente diferentes. Essa variedade de *hardware* não se restringe apenas aos tipos de computadores, se aplica também aos seguintes aspectos:

- Redes;
- Sistemas operacionais;
- Linguagens de programação;
- Implementação de diferentes desenvolvedores.

A internet, por exemplo, embora composta por vários tipos de redes, tem suas diferenças mascaradas pelo fato de que todos os computadores ligados a elas utilizam protocolos diferentes para se comunicar. Por exemplo, um computador que possui uma placa *ethernet* tem uma implementação dos protocolos Internet, específicos para a rede a qual faz parte. Enquanto um computador em um tipo diferente de rede, tem uma implementação dos protocolos Internet para essa rede. Dessa forma, é possível a troca de informações entre essas redes sem que o usuário perceba.

Todas essas variedades e diferenças que podem existir em um sistema distribuído, geralmente são mascaradas para que não possam ser percebidas pelos usuários, dessa característica surge outro desafio a ser considerado, a transparência.

2.1.2 Transparência

A transparência em um sistema distribuído é uma consequência direta da definição de Tanenbaum, pois é justamente a ocultação, para os usuários e aplicações, da separação física dos recursos e processos dentre os computadores do sistema, i.e., os usuários têm a impressão de estarem utilizando o sistema em um único computador. Existem alguns tipos de transparência:

- Acesso - Recursos locais e remotos podem ser acessados da mesma maneira;
- Localização - Os recursos podem ser acessados sem conhecimento da sua localização física;
- Concorrência - Um recurso pode ser compartilhado por vários usuários concorrentes;
- Replicação - Oculta que um recurso é replicado, i.e., os usuários e programadores das aplicações, não possuem conhecimento das réplicas dos recursos entre os computadores do sistema;
- Falhas - Permite a ocultação de falhas e recuperação de um recurso possibilitando que usuários e programas concluam suas tarefas;
- Desempenho - Permite que o sistema seja reconfigurado para melhorar o desempenho a medida que as cargas variam;
- Escalabilidade - Permite adição de mais recursos no sistema sem alterar sua estrutura.

2.2 Sistemas Abertos

Um sistema distribuído é considerado aberto, principalmente pelo grau com que novos serviços de compartilhamento de recursos podem ser adicionados e disponibilizados para uso por uma variedade de programas clientes.

Essa característica é adquirida através de regras padronizadas que descrevem a sintaxe e semântica desses serviços e são publicadas através de documentos e especificações. Essas publicações permitem a extensão desses sistemas por diferentes desenvolvedores. É possível ampliá-los em nível de *hardware*, pela adição de computadores em uma rede, e em nível de *software*, pela introdução de novos serviços ou pela reimplementação dos antigos, permitindo as aplicações compartilharem recursos. Outra vantagem para sistemas abertos além da extensibilidade é a independência de fornecedores individuais. As RFC's, ou *Request For Comments*, é um exemplo dessas especificações que são publicadas através de documentos.

Os protocolos de Internet, por exemplo, são especificados através das RFC's. Através dessas especificações, que os desenvolvedores das diversas redes que compõem a Internet, seguem padrões que tornam possível a comunicação entre redes diferentes.

2.3 Escalabilidade

Sistemas distribuídos existem em diversas escalas, desde uma pequena intranet até a Internet e funcionam de forma eficiente e eficaz.

Um sistema que se mantém eficiente quando há um aumento significativo no número de recursos e usuários é denominado escalável. Ao ser ampliado, um sistema apresenta muitas vezes uma perda no seu desempenho. Por esse motivo, quando existe a necessidade de adicionar novos recursos é preciso resolver alguns problemas.

Deve ser possível ampliar um sistema, a um custo razoável, de acordo com o aumento da demanda por um recurso. Em uma intranet, por exemplo, deve ser possível adicionar novos servidores de arquivos, de acordo com o aumento de acesso aos arquivos. Essa adição de novos recursos, visa evitar gargalo de desempenho no sistema, caso um único servidor de arquivos seja responsável por tratar todas as requisições. Evitar os gargalos de desempenhos, quando recursos compartilhados são acessados com muita frequência (como acontece com as páginas Web), deve ser possível. O uso de cache e de replicação, melhora o desempenho de recursos que são pesadamente utilizados [13].

Existem vários outros problemas que devem ser considerados no projeto de um sistema distribuído escalável, mas de forma resumida, o sistema deve se manter eficiente quando alteramos sua escala. As técnicas que tem obtido sucesso para tratar os problemas de escala, incluem principalmente dados replicados, uso de cache e distribuição de vários servidores para manipular as tarefas mais executadas. Permitindo que várias tarefas do mesmo tipo sejam executadas de maneira concorrente [13], além da comunicação assíncrona [26].

2.4 Concorrência

Uma característica muito importante dos sistemas distribuídos, e que é inclusive uma das principais motivações para seu desenvolvimento e utilização, é o compartilhamento de recursos. Esses recursos podem variar como recursos de *hardware* (como impressoras), recursos de dados (como arquivos) e recursos com funcionalidades mais específicas (como os mecanismos de busca). O processo que gerencia um recurso compartilhado poderia aceitar e tratar um pedido de cada cliente por vez. Contudo, essa estratégia limita o desempenho do tratamento de cada pedido. Portanto, os serviços e aplicações geralmente permitem que vários pedidos de clientes sejam processados concorrentemente. A existência desses acessos concorrentes sobre recursos compartilhados, implica em alguns problemas que devem ser tratados, para evitar o mau funcionamento do sistema ou até a sua inviabilização.

Em suma, qualquer objeto que represente um recurso compartilhado em um sistema distribuído, deve ser responsável por garantir que ele opere corretamente em um ambiente concorrente. Isso se aplica não apenas aos servidores, mas também aos objetos nas aplicações. Portanto, qualquer programador que implemente um objeto que não foi destinado para o uso em um sistema distribuído, deve fazer o que for necessário para garantir que, em um ambiente concorrente, ele não assuma resultados inconsistentes.

Para que um objeto mantenha coerência em um ambiente concorrente, suas operações devem ser sincronizadas de tal maneira que seus dados permaneçam consistentes. Isso

pode ser obtido por meio de técnicas padrões, como semáforos [28], que estão disponíveis na maioria dos sistemas operacionais.

2.5 Tratamento de Falhas

Os sistemas distribuídos estão sujeitos a diversas falhas que podem ocorrer em qualquer uma de suas partes (redes, computadores e programas). Se essas falhas não forem tratadas o sistema pode produzir resultados incorretos ou pode parar antes de ter concluído a requisição solicitada.

A causa de um erro é uma falha (*fault*). O funcionamento inadequado de um sistema geralmente é identificado pelos usuários através da exibição de um erro, que é uma parte do estado do sistema, e por isso pode ser avaliado e observado pelos usuários. Desta forma, é através dos erros apresentados que o usuário pode identificar se o sistema possuí uma falha. Apesar de uma falha ter o potencial de gerar erros, ela pode não gerar erro algum durante o período de sua observação. As falhas podem ser classificadas como transientes, intermitente e permanente.

Falhas transientes ocorrem uma vez e depois desaparecem. São aquelas de duração limitada, causadas por mau funcionamento temporário ou por alguma interferência externa. Um pássaro que voa pelo feixe de um transmissor de microondas, pode causar perda de bits em alguma rede. Se a temporização de uma transmissão se esgotar e tentarmos executá-la novamente, ela provavelmente funcionará desta segunda vez [28].

Uma falha intermitente é aquela que ocorre repetidamente por curtos intervalos de tempo, ocorre e desaparece várias vezes por vontade própria. Um conector com o contato frouxo geralmente causa uma falha intermitente.

Já as falhas permanentes são aquelas que quando um componente falha, a falha somente é corrigida quando esse componente é substituído.

Ao contrário do que acontece nos sistemas não distribuídos, em que a falha de um componente quase sempre é total, no sentido que afeta todos os componentes e pode facilmente fazer o sistema inteiro cair, nos sistemas distribuídos as falhas devem ser parciais. Quando algum componente falha, outros componentes podem ser afetados, mas a maioria deve permanecer em funcionamento, garantindo que o sistema continue operando conforme sua especificação.

Ao perceber o mau funcionamento de um servidor em um sistema distribuído, muitas vezes não significa que a falha que está ocasionando o problema, está exatamente nesse servidor. Isso porque, tal servidor pode depender de outros servidores para prestar seus serviços, pode ser que a causa de um erro tenha sido provocada em outro lugar. Essas relações de dependência aparecem constantemente em sistemas distribuídos. Uma definição interessante de sistemas distribuídos que expõe essas relações é:

> "Você sabe que está diante de um sistema distribuído quando um computador do qual você nunca ouviu falar para de funcionar e impede você de terminar o seu trabalho."
>
> Leslie Lamport

Um exemplo dessa relação de dependência, pode ser exemplificado em um sistema de arquivos projetado para oferecer alta disponibilidade. Um disco que está falhando, pode

dificultar a vida de um servidor desse sistema, se tal servidor fizer parte de um banco de dados distribuído. O funcionamento adequado de todo o banco de dados pode estar em jogo, pois parte dos seus dados podem ter sido afetadas e estarem indisponíveis [28].

Existem diferentes tipos de falhas que são classificadas de acordo com uma taxonomia proposta por Hadzilacos e Toueg [1994]. Tal divisão é apresentada sob os títulos falhas por omissão, falhas de temporização, falhas de respostas e falhas bizantinas (falhas arbitrárias).

As falhas por omissão, se referem aos casos em que um processo ou canal de comunicação, deixa de executar as ações que deveria.

As falhas de temporização, ocorrem quando a resposta se encontra fora de um intervalo de tempo real especificado. O mais comum é que um servidor responda tarde demais, quando então se diz que ocorreu uma falha de desempenho.

Outro tipo de falha é a falha de resposta, na qual a resposta do servidor é simplesmente incorreta.

Já as falhas bizantinas, também conhecidas por falhas arbitrárias, é o tipo de falha mais grave em um sistema distribuído, pois é difícil de prever quais partes do sistema irão afetar e quando. Uma falha bizantina ocorre quando o sistema age de forma arbitrária a qualquer momento. Um servidor, por exemplo, pode estar produzindo saídas que nunca deveria ter produzido, mas que não podem ser identificadas como incorretas. Uma situação ainda pior é quando um servidor age de maneira maliciosa, ou seja, um processo bizantino pode, por exemplo, tentar assumir a identidade de outro, enviar mensagens com valores incorretos, duplicar mensagens ou simplesmente não enviar as mensagens necessárias.

A construção de sistemas confiáveis, está diretamente relacionada com o controle de falhas. Pode-se fazer uma distinção entre evitar, remover e prever falhas [4]. Mas para a finalidade desse projeto, que engloba o desenvolvimento de um sistema de arquivos distribuído tolerante a falhas, o mais importante é a tolerância a falhas, o que significa que um sistema pode prover seus serviços mesmo na presença de falhas.

Existem algumas técnicas que são utilizadas para dar tratamento as falhas [28]:

- Detecção de falhas: algumas falhas podem ser detectadas. Por exemplo, somas de verificação podem ser usadas para detectar dados corrompidos em uma mensagem ou arquivo. É difícil, ou mesmo impossível, detectar algumas outras falhas, como um servidor remoto danificado na internet. O desafio é gerenciar a ocorrência de falhas que não podem ser detectadas, mas que podem ser suspeitas;

- Mascaramento de falhas: Existe a possibilidade de ocultar ou minimizar o impacto no sistema de algumas falhas, que podem ser detectadas. Como exemplo de ocultação de falhas, mensagens podem ser retransmitidas. Simplesmente eliminar uma mensagem corrompida é um exemplo de como tornar uma falha menos grave - ela pode ser retransmitida;

- Tolerância a falhas: Pode ser definida como a habilidade do sistema de apresentar um comportamento muito bem definido na presença de falhas ativas, ou seja, o sistema mantém seu funcionamento mesmo na ocorrência dessas falhas. Não seria prático, no caso da internet, que seus serviços apresentam várias falhas, tentar detectar e mascarar tudo que possa ocorrer em uma rede dessas, com tantos componentes. Um navegador, por exemplo, quando não consegue contatar um servidor Web, não

faz o usuário esperar indefinidamente, enquanto contínua tentando - ele informa ao usuário sobre o problema, deixando-o livre para tentar novamente;

- Recuperação de falhas: A recuperação envolve projetar o *software*, de modo que o estado dos dados permanentes, possa ser recuperado após a falha de um servidor;
- Redundância: os SD's podem se tornar tolerantes a falhas, utilizando componentes redundantes em sua composição. Um exemplo dessa redundância é a replicação de um banco de dados em vários servidores, visando garantir que os dados permaneçam acessíveis após a falha de qualquer servidor. Outro exemplo, é a existência de pelo menos duas rotas diferentes entre dois roteadores quaisquer na Internet;

2.6 Segurança

Projetar e manter um sistema distribuído seguro é uma tarefa difícil, diante da quantidade de ameaças que o mesmo pode encontrar. Manter seguro os dados que são processados e armazenados nesse tipo de sistema é um dos maiores desafios. Geralmente esse tipo de sistema é formado por muitas redes e componentes, o que dificulta manter o sistema seguro em todos os pontos. Muitos recursos de informação que se tornam disponíveis e são mantidos em sistemas distribuídos, têm um alto valor intrínseco para seus usuários. Portanto, sua segurança é de considerável importância.

A segurança nos sistemas computacionais é composta pelos atributos de confidencialidade, integridade e disponibilidade. Manter a confidencialidade no sistema significa não permitir a divulgação não autorizada das informações, i.e., somente as partes autorizadas podem acessar determinadas informações. A integridade consiste na realização de alterações nos ativos do sistema somente por partes autorizadas, i.e., alterações indevidas devem ser detectáveis e recuperáveis. Já a disponibilidade pode ser caracterizada como a capacidade do sistema oferecer o serviço correto sempre que solicitado.

Projetar ou tornar um sistema distribuído totalmente seguro, considerando todos as ameaças e implementando cada um desses atributos é uma tarefa desafiadora. Isso porque a implementação de técnicas de segurança, geralmente tem um impacto significativo no desempenho do sistema. Por isso, deve ser realizado uma análise balanceando a necessidade de implementação de determinada técnica de segurança, com o seu possível impacto no desempenho do sistema.

Capítulo 3

Sistemas de Arquivos Distribuídos

Como já foi apresentado no Capítulo 2, um dos principais objetivos dos sistemas distribuídos é o compartilhamento de recursos. O compartilhamento de informações armazenadas talvez seja o aspecto mais importante dos recursos distribuídos e essas informações geralmente são armazenadas em forma de arquivos, daí surge a necessidade do desenvolvimento de sistemas de arquivos distribuídos.

Inicialmente, os sistemas de arquivos foram desenvolvidos para sistemas de computadores centralizados e desktop's como um recurso do sistema operacional, que fornece uma interface de programação conveniente para armazenamento em disco. Posteriormente, eles adquiriram características como controle de acesso e mecanismos de proteção de arquivos, que os tornaram úteis para o compartilhamento de dados e programas.

Um serviço de arquivo bem projetado dá acesso a arquivos armazenados em um servidor com desempenho e confiabilidade, semelhantes a arquivos armazenados em discos locais. Permite que programas armazenem e acessem arquivos remotos exatamente como se fosse locais, possibilitando que os usuários acessem seus arquivos a partir de qualquer computador da rede.

3.1 Sistemas de arquivos

Os sistemas de arquivos têm a função de organizar, armazenar, recuperar, atribuir nomes, compartilhamento e proteção de arquivos. Fornecem uma interface de programação que oferece uma abstração dos arquivos, ou seja, permitem que os programadores não tenham preocupação com os detalhes da alocação e armazenamento físico.

Os arquivos são formados por dados e metadados. Os dados consistem em uma sequência de elementos (geralmente, bytes – 8 bits), acessíveis pelas operações de leitura e escrita de qualquer parte dessa sequência. Já os metadados são mantidos como um registro único, contendo informações como o tamanho do arquivo, carimbo de tempo, tipo de arquivo, identidade de proprietário e listas de controle de acesso. A Tabela 3.1 mostra a estrutura básica dos metadados.

Recursos para criação, atribuição de nomes e exclusão de arquivos, compõem o projeto de sistema de arquivos. Esses recursos possuem a a finalidade de armazenar e gerenciar um grande número de arquivos.

Tamanho do arquivo
Horário de criação
Horário de acesso (leitura)
Horário de modificação (escrita)
Horário de alteração de atributo
Contagem de referência
Proprietário
Tipo de arquivo
Lista de controle de acesso

Tabela 3.1: Estrutura básica dos metadados

O controle de acesso aos arquivos é uma das responsabilidades do sistema de arquivos e é implementado restringindo o acesso de acordo com as autorizações dos usuários e com o tipo de acesso solicitado (leitura, atualizações, execução, etc.).

3.1.1 Requisitos do Sistema de Arquivos Distribuído

Conforme visto no Capítulo 2, os sistemas distribuídos apresentam desafios para seu projeto e desenvolvimento e isso não é diferente nos sistemas de arquivos distribuídos. Além da solução desses desafios, existem outros requisitos a serem considerados no projeto dos sistemas de arquivos distribuídos. Os principais requisitos são [13]:

- Transparência: O serviço de arquivos deve suportar muitos requisitos de transparência dos sistemas distribuídos, citados no Capítulo 2. As formas de transparências seguintes são parcialmente ou totalmente tratadas pelos serviços de arquivos atuais:
  - Transparência de acesso: os programas clientes não devem conhecer a distribuição de arquivos. Um único conjunto de operações é fornecido para acesso a arquivos locais e remotos.
  - Transparência de localização: os programas clientes devem ver um espaço de nomes de arquivos uniforme. Os arquivos, ou grupo de arquivos, podem ser deslocados de um servidor a outro sem alteração de seus nomes de caminho, e os programas de usuários devem ver o mesmo espaço de nomes onde quer que sejam executados.
  - Transparência de mobilidade: nem os programas clientes, nem as tabelas de administração de sistema nos computadores clientes precisam ser alterados quando os arquivos são movidos. Isso permite a mobilidade dos arquivos. Conjuntos ou volumes de arquivos podem ser movidos, ou pelos administradores de sistema ou automaticamente.
  - Transparência de desempenho: os programas clientes devem continuar a funcionar satisfatoriamente enquanto a carga sobre o serviço varia dentro de um intervalo especificado.
  - Transparência de mudança de escala: o serviço pode ser expandido de forma gradual, para lidar com uma ampla variedade de cargas e tamanhos de rede.

- Atualizações concorrentes de arquivos: quando um usuário altera um arquivo isso não deve interferir na operação de outros clientes que estejam acessando, ou alterando, o mesmo arquivo simultaneamente. Em muitos programas, a necessidade de controle de concorrência para acesso a dados compartilhados é amplamente aceita, e são conhecidas técnicas para sua implementação, mas elas são dispendiosas. A maior parte dos serviços de arquivos atuais segue os padrões UNIX [5] modernos, fornecendo travamento (*locking*) em nível de arquivos ou em nível de registro.

- Replicação de arquivos: Significa que várias cópias do mesmo arquivo são distribuídas em diferentes locais. As duas principais vantagens dessa replicação são 1) permite que vários servidores compartilhem a carga do fornecimento de um serviço para clientes que acessam o mesmo conjunto de arquivos, melhorando a escalabilidade do serviço e 2) melhora a tolerância a falhas, permitindo que, em caso de falhas, os clientes localizem outro servidor que contenha uma cópia do arquivo.

- Heterogeneidade do *hardware* e do sistema operacional: As interfaces de serviço devem ser definidas de modo que o *software* cliente e servidor possam ser implementados para diferentes sistemas operacionais e computadores.

- Tolerância a Falhas: é a capacidade do sistema de arquivos distribuídos continuar em funcionamento mesmo diante da presença de falhas. Os servidores podem ser sem estado (*stateless*), para que após uma falha o serviço possa ser reiniciado e restaurado sem necessidade de recuperar um estado anterior. A tolerância a falha de queda (*crash*) exige replicação de arquivos, a qual é difícil de obter.

- Consistência: Os sistemas de arquivos convencionais oferecem semântica de atualização de cópia única (*one-copy*). Isso se refere a um modelo de acesso concorrente a arquivos, no qual o conteúdo do arquivo visto por todos os processos que estão acessando, ou atualizando determinado arquivo, é aquele que eles veriam se existissem apenas uma cópia do conteúdo do arquivo. Quando os arquivos são replicados, ou armazenados em *cache*, em diferentes sites, há um atraso inevitável na propagação das modificações feitas em um site para os outros sites que contêm cópias, e isso pode resultar em certo desvio da semântica de cópia única.

- Segurança: O projeto de sistema de arquivos distribuído deve conter políticas e mecanismos de segurança que garantam a disponibilidade, integridade e confidencialidade desse sistema, atributos que compõem a segurança em sistemas da informação. Nos sistemas de arquivos, o controle de acesso é baseado no uso de listas de controle de acesso. Nos sistemas de arquivos distribuídos, há necessidade de autenticar as requisições dos clientes para que o controle de acesso no servidor seja baseado nas identidades corretas de usuários e para proteger o conteúdo das mensagens de requisição-resposta com assinaturas digitais e criptografia de dados quando necessário.

- Eficiência: O desempenho de sistemas de arquivos distribuídos deve ser comparável ao de sistemas de arquivos convencionais.

3.2 Tolerância a Falhas

Um sistema de arquivos distribuído tolerante a falhas implementa técnicas que garantem o funcionamento correto do sistema mesmo na ocorrência de falhas. A tolerância a falhas [3] é realizada através de detecção de erros e recuperação do sistema. A Figura 3.1 apresenta as técnicas envolvidas na tolerância a falhas. Essas técnicas garantem funcionamento correto do sistema mesmo na ocorrência de falhas e são todas baseadas em redundância, exigindo componentes adicionais ou algoritmos especiais.

Geralmente, tratamento de falhas é seguido por uma manutenção corretiva, que tem como objetivo remover as falhas que são isoladas pelo tratamento de falhas, ou seja, o que diferencia a tolerância a falhas de manutenção é que a manutenção requer a participação de um agente externo.

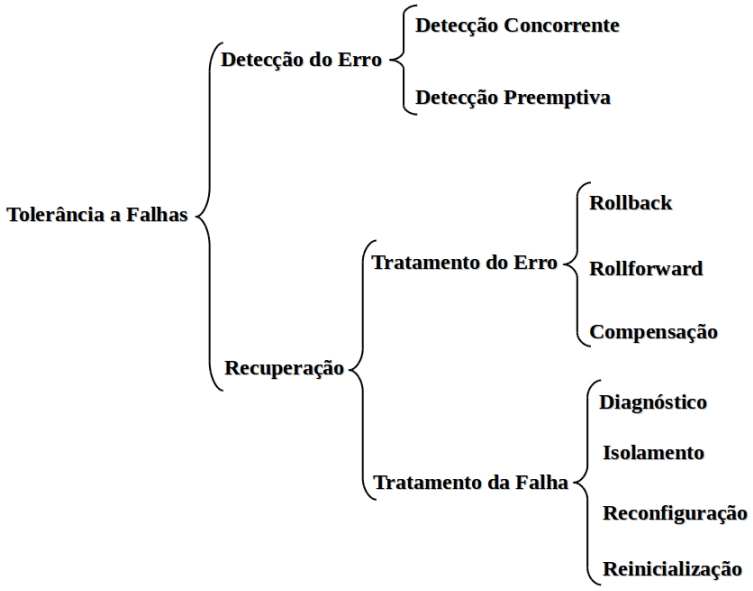

Figura 3.1: Técnicas de tolerância a falhas. Adaptado de [4]

A detecção de erros subdivide-se em detecção concorrente e detecção preemptiva, e corresponde a identificação da presença de um erro. A detecção concorrente ocorre durante a entrega normal do serviço. A detecção preemptiva ocorre quando a entrega normal do serviço está suspensa checando o sistema por erros latentes e falhas dormentes. Na recuperação, um estado do sistema que contenha um ou mais erros e possíveis falhas é transformado em um estado sem erros detectados e sem falhas que possam ser ativadas de novo. Subdivide-se em: tratamento de erros e tratamento de falhas.

Tratamento de falhas evita as falhas de serem ativadas outra vez, e subdivide-se em:

- Reinicialização: testa, atualiza e registra uma nova configuração, atualizando tabelas e também registros do sistema.
- Reconfiguração: troca para componentes reservas ou redistribui tarefas dentre os componentes não defeituosos.

- Isolamento: efetua uma exclusão física ou lógica de componentes defeituosos, ou seja, os torna dormentes.
- Diagnóstico: identifica e registra as causas dos erros, em termos tanto de suas localizações quanto de seus tipos.

Tratamento de erros elimina erros do estado do sistema, e subdivide-se em:

- Rollforward: um estado sem erros detectados é o novo estado.
- Rollback: leva o sistema para um estado anterior a ocorrência dos erros.
- Compensação: o estado errôneo contém suficiente redundância para permitir que o erro seja mascarado.

Considerando as formas de redundância de *hardware* e *software*, estas se baseiam na replicação de partes de componentes de um sistema, podendo também considerar até o sistema como um todo. Réplicas podem ser idênticas em sua constituição ou não, mas sempre possuem a mesma função, a capacidade de permitir a tolerância de falhas permanentes.

Exemplo de redundância na estrutura baseada em *software* é a técnica conhecida como *N-version programming*. Na abordagem é considerada a utilização de duas ou mais versões de um só algoritmo, que é programado de forma independente, comparando suas saídas, das quais é escolhido o resultado correto. A técnica em *hardware* é conhecida como *N-modular redundancy*, em que se utilizam cópias de um mesmo módulo de *hardware*.

Nos sistemas de arquivos distribuídos, a tolerância a falhas é obtida, principalmente, através da replicação de dados que é a distribuição de cópias dos dados em vários computadores. A replicação é o segredo da eficácia dos sistemas distribuídos, pois pode fornecer um melhor desempenho, alta disponibilidade e tolerância a falhas. A replicação é amplamente usada como, por exemplo, no armazenamento de recursos de servidores web, na cache dos navegadores e em servidores proxy, pois os dados mantidos no cache e no proxy são réplicas uns dos outros. O serviço de atribuição de nomes DNS mantém cópias de mapeamentos entre nomes e atributos dos computadores e conta-se com ele para o acesso diário aos serviços pela Internet. A replicação por máquina de estados exige o determinismo de réplicas: todas as réplicas partem de um mesmo estado inicial e executam a mesma sequência de operações (na mesma ordem), chegando a um mesmo estado final. A replicação é uma técnica para melhorar serviços. As motivações para a replicação são 1) melhorar o desempenho de um serviço, 2) aumentar sua disponibilidade ou 3) torná-lo tolerante à falha.

Os métodos baseados em redundância no tempo, são caracterizados pela repetição da mesma atividade uma ou mais vezes. O motivo da repetição se baseia no fato de que a causa do problema é de natureza temporal. Um exemplo clássico de redundância no tempo é a retransmissão de mensagens que ocorre em protocolos de comunicação. Tomando o protocolo TCP (da arquitetura TCP/IP) como exemplo, na utilização deste protocolo é efetuada uma retransmissão de segmentos TCP todas as vezes que o remetente da mensagem deixa de receber a confirmação do recebimento pelo destinatário.

3.2.1 Tolerância a falhas no FARSITE

FARSITE [1] é um sistema de arquivos seguro e escalável, que logicamente funciona como um servidor de arquivos centralizado, mas é fisicamente distribuído entre um conjunto de computadores não confiáveis.

É um sistema de arquivos distribuído que não assume a confiança mútua entre os computadores cliente em que é executado. Logicamente, o sistema funciona como um servidor de arquivos central, mas fisicamente, não há nenhuma máquina do servidor central. Em vez disso, um grupo de computadores cliente estabelecem de forma colaborativa um servidor de arquivos virtual, que pode ser acessado por qualquer um dos clientes.

O sistema fornece um espaço de nomes global para os arquivos, acesso à localização-transparente para os arquivos privados e arquivos públicos compartilhados, e maior confiabilidade em relação ao armazenamento de arquivos em um desktop. Ele faz isso através da distribuição de várias réplicas criptografadas de cada arquivo entre um conjunto de máquinas clientes. Os arquivos são referenciados através de uma estrutura hierárquica de diretórios que é mantido por um serviço de diretório distribuído.

Cada máquina no FARSITE pode realizar três funções: É um cliente, membro de um grupo de diretório, e um hospedeiro de arquivo, mas inicialmente vamos ignorar o último deles. Um cliente é uma máquina que interage diretamente com um usuário. Um grupo de diretório é um conjunto de máquinas que gerenciam coletivamente informações do arquivo usando um protocolo tolerante a falhas bizantinas [12]: Cada membro do grupo armazena uma réplica da informação, e como o grupo recebe pedidos de clientes, cada membro processa essas solicitações de forma determinística, atualiza sua réplica, e envia respostas para o cliente. O protocolo Bizantino garante a consistência dos dados, desde que menos de um terço das máquinas se comportam mal.

Considere um sistema que inclui vários clientes e um grupo de diretório. Para o momento, imagina que o grupo diretório gerencia todos os dados do sistema de arquivos e metadados, armazena eles de forma redundante em todas as máquinas do grupo. Quando um cliente deseja ler um arquivo, ele envia uma mensagem para o grupo de diretórios, que responde com o conteúdo do arquivo solicitado. Se o cliente atualiza o arquivo, ele envia a atualização para o grupo de diretório. Se outro cliente tenta abrir o arquivo enquanto o primeiro cliente está com ele aberto, o grupo de diretório avalia a semântica de compartilhamento especificada de cada solicitação feita por cada cliente, para determinar se a concede segundo acesso ao outro cliente.

O sistema FARSITE implementa a tolerância a falhas principalmente através da replicação. O conjunto de metadados dos arquivos é replicado entre os membros de um grupo de diretório e de dados de arquivos é replicado em vários hospedeiros de arquivos. Grupos de diretório empregam replicação de tolerância a falhas bizantinas e os hospedeiros de arquivos empregam replicação simples [10].

Para o armazenamento redundante de dados dos arquivo, o FARSITE poderia ter usado um esquema de *erasure coding* [10], em vez de replicação simples, foi escolhido o último, em parte, porque é mais simples e, em parte, porque existiam preocupações sobre a latência adicional introduzida pelo fragmento de remontagem de dados com *erasure coding* durante as leituras dos arquivos.

No que diz respeito à confiabilidade, replicação contra a morte permanente de máquinas individuais, incluindo falhas de perda de dados e desativação explícita do usuário. No que diz respeito à disponibilidade, replicação contra a inacessibilidade transitória de

máquinas individuais, incluindo falhas no sistema, partições de rede e desligamentos explícitos. Em um grupo de diretório de R membros, os metadados são preservados e acessíveis, se não mais do que *(R - 1) / 3* das máquinas se tornarem inacessíveis. Para arquivos replicados em F hospedeiros de arquivos, os dados de arquivo são preservados e acessíveis se pelo menos um hospedeiro de arquivo permanece acessível.

Na concepção FARSITE, quando um computador não está disponível por um período prolongado de tempo, as suas funções migram para uma ou mais outras máquinas, utilizando as outras réplicas dos dados de arquivos e metadados de diretório para regenerar os dados e metadados sobre as máquinas de substituição. Assim, os dados são perdidos para sempre só se muitas máquinas falham dentro de uma janela de tempo muito pequena para permitir a regeneração.

Como o volume de dados do diretório é muito menor do que o de dados de arquivo, migração de diretórios é realizada de forma mais agressiva do que a migração de hospedeiros de arquivos. Sempre que um membro do grupo está fora ou inacessível, mesmo para um curto período de tempo, os outros membros do grupo escolhem um substituto aleatoriamente do conjunto de máquinas acessíveis que eles conhecem. Já que máquinas de baixa disponibilidade estão por definições ativas por uma fração menor de tempo do que as máquinas de alta disponibilidade, são mais propensas a ter seu estado migrado para outra máquina e menos propensas a ser um alvo acessível para a migração de outra máquina.

3.3 Segurança em Sistemas Distribuídos

A segurança é uma das questões mais importantes em sistemas distribuídos. Quando os dados são distribuídos através de múltiplas redes ou informação é transferida através de redes públicas, torna-se vulneráveis a ataques por elementos maliciosos. Similarmente outros recursos de computação como processadores, dispositivos de armazenamento, redes, etc., também podem ser atacados por hackers.

Dependabilidade(confiança) é introduzido pela primeira vez como um conceito que engloba os atributos habituais de confiabilidade, disponibilidade, segurança, integridade, capacidade de manutenção, etc. A consideração de segurança traz preocupações de confidencialidade, além de disponibilidade e integridade. As definições básicas são então comentadas e complementada por definições adicionais.

3.3.1 Os Conceitos Básicos

Nesta seção, apresentamos um conjunto básico de definições que serão utilizados ao longo de toda a discussão sobre a taxonomia de computação confiável e seguro. As definições são gerais o suficiente para cobrir toda a gama de sistemas de computação e comunicação, a partir de portas lógicas individuais a redes de computadores com operadores humanos e usuários. No que se segue, vamos nos concentrar principalmente em sistemas de computação e comunicação, mas as nossas definições destinam-se também em grande parte a ser de relevância para os sistemas baseados em computador, ou seja, sistemas que também abrangem os seres humanos e organizações que fornecem o ambiente imediato da computação e sistemas de comunicação de interesse.

Sistema, Comportamento, Estrutura e Serviço

Um sistema em nossa taxonomia é uma entidade que interage com outras entidades, ou seja, outros sistemas, incluindo *hardware*, *software*, os seres humanos e o mundo físico com seus fenômenos naturais. Estes outros sistemas são os ambientes do sistema dado. O limite do sistema é a fronteira comum entre o sistema e seu ambiente. Sistemas de computação e comunicação são caracterizados por propriedades fundamentais: funcionalidade, desempenho, confiabilidade e segurança, e custo. Outras propriedades importantes do sistema que afetam a confiabilidade e segurança incluem usabilidade e gerenciabilidade [31]. A função de um tal sistema é que o sistema se destina a fazer e é descrito pela especificação funcional em termos de funcionalidade e desempenho. O comportamento de um sistema é que o sistema faz a executar a sua função e é descrita por uma sequência de estados. O estado total de um dado sistema é o conjunto dos seguintes estados: computação, comunicação, informação armazenada, interconexão, e condição física.

A estrutura de um sistema é o que lhe permite gerar o comportamento. De um ponto de vista estrutural, um sistema é composto por um conjunto de componentes ligados entre si de modo a interagir, onde cada componente é um outro sistema, etc. A recursão para quando um componente é considerado atômica: Qualquer outra estrutura interna não pode ser discernida, ou não é de interesse e podem ser ignorados. Consequentemente, o estado total de um sistema é o conjunto dos estados (externos) de seus componentes atômicos.

O serviço prestado por um sistema (em seu papel de provedor) é o seu comportamento como é percebido pelo seu usuário (s); um usuário é outro sistema que recebe serviço do provedor. A parte da fronteira do sistema do prestador de onde a prestação de serviços ocorre é interface de serviço do provedor. A parte do estado total do provedor que é perceptível na interface de serviço é o seu estado externo; a parte restante é seu estado interno. O serviço prestado é uma sequência de estados externos do provedor. Nota-se que um sistema pode sequencialmente ou simultaneamente ser um fornecedor e um utilizador no que diz respeito a outro sistema, isto é, para fornecer um serviço e receber o serviço de que outro sistema. A interface do utilizador a que o utilizador recebe o serviço é interface de uso.

Nós temos até agora usado o singular para a função e serviço. Um sistema implementa geralmente mais do que uma função, e proporciona mais de um serviço. Função e serviço pode ser assim visto como composta de itens de função e de itens de serviço. Por uma questão de simplicidade, vamos simplesmente usar o plural funções, serviços de quando é necessário distinguir vários itens de função ou serviço.

As Ameaças à Confiança e Segurança: Falhas, Erros e Defeitos

Serviço correto é entregue quando o serviço implementa a função de sistema. Uma falha no serviço é um evento que ocorre quando o serviço prestado se desvia do serviço correto. Um serviço falhar (failure) significa que ele não está em conformidade com a especificação funcional, ou porque esta especificação não descreve adequadamente a função do sistema. Uma falha no serviço é uma transição de serviço correto para o serviço incorreto, ou seja, a não implementação da função do sistema. O prazo de entrega do serviço incorreto é uma interrupção do serviço. A transição de serviço incorreto para corrigir serviço é um serviço de restauração. O desvio do serviço correto pode assumir

diferentes formas que são chamados de modos de falha do serviço e são classificados de acordo com a severidade de falha.

Uma vez que um serviço é uma sequência de estados externos do sistema, uma falha no serviço significa que, pelo menos, um (ou mais) estado externo do sistema se desviar do estado de serviço !!br0ken!! O desvio é chamado um erro. A causa julgada ou hipótese de um erro é chamado uma falha. Falha pode ser interna ou externa de um sistema. A presença anterior de uma vulnerabilidade, ou seja, uma falha interna que permite que uma falha externa para prejudicar o sistema, é necessário para um defeito externo para causar um erro e possivelmente insuficiência subsequente (s). Na maioria dos casos, um primeiro defeito provoca um erro no estado de um serviço componente que é uma parte do estado interno do sistema e do estado externo não é imediatamente afetado.

Por este motivo, a definição de um erro, é a parte do estado total do sistema, que pode levar à sua falha no serviço subsequente. É importante notar que muitos erros não atingem o estado externo do sistema e causam uma falha. Um defeito está ativo quando se provoca um erro, caso contrário ele está dormente.

Quando a especificação funcional de um sistema inclui um conjunto de várias funções, a falha de um ou mais dos serviços que desempenham as funções podem deixar o sistema em um modo degradado que ainda apresenta um subconjunto dos serviços necessários para o utilizador. A especificação pode identificar vários desses modos, por exemplo, serviço lento, serviço limitado, serviços de emergência, etc. Aqui, dizemos que o sistema sofreu uma falha parcial de sua funcionalidade ou desempenho.

Confiabilidade, Segurança, e seus Atributos

A definição original de segurança é a capacidade de fornecer um serviço razoavelmente confiável. Esta definição realça a necessidade de justificação da confiança. A definição alternativa que fornece o critério para decidir se o serviço é confiável é a confiabilidade de um sistema que é a capacidade de evitar falhas de serviço que são mais frequentes e mais graves do que é aceitável.

É costume dizer que a dependabilidade de um sistema deve ser suficiente para a dependência de serem colocados no sistema. A dependência de um sistema A no sistema B, assim, representa o grau de dependabilidade do sistema que A é (ou seriam) afetada por esse sistema de B. O conceito de dependência considera que a confiança, a qual pode convenientemente ser definida como a dependência aceita.

Como desenvolvido ao longo das últimas três décadas, dependabilidade é um conceito integrado que engloba os seguintes atributos:

- Disponibilidade: prontidão para serviço correto;
- Confiabilidade: a continuidade do serviço correto;
- Segurança: ausência de consequências catastróficas sobre o usuário(s) e o meio ambiente;
- Integridade: ausência de alterações inadequadas do sistema;
- Manutenção: a capacidade de sofrer modificações e reparos.

Ao abordar a segurança, um atributo adicional tem grande destaque, confidencialidade, ou seja, a ausência de divulgação não autorizada de informações. Segurança é um composto dos atributos de confidencialidade, integridade e disponibilidade, exigindo a existência simultânea de 1) disponibilidade para ações autorizados somente, 2) de confidencialidade, e 3) a integridade.

A especificação de confiabilidade e segurança de um sistema deve incluir os requisitos para os atributos em termos de frequência e gravidade aceitável de falhas de serviço para classes específicas de falhas e um determinado ambiente de utilização. Um ou mais atributos podem não ser necessários em todos as fases de desenvolvimento do sistema.

Os Meios Para Alcançar a Confiança e a Segurança

Ao longo dos últimos 50 anos, muitos meios foram desenvolvidos para atingir os vários atributos de dependabilidade e de segurança. Esses meios podem ser agrupados em quatro grandes categorias:

- Prevenção de falhas, para prevenir a ocorrência ou introdução de falhas;
- A tolerância a falhas, evitar falhas de serviço na presença de falhas;
- Falha de remoção, para reduzir o número e gravidade de falhas;
- Previsão de falha, para estimar o número atual, a incidência futura, e as prováveis consequências de falhas;
- Prevenção a falhas e tolerância a falhas, como objetivo de proporcionar a capacidade de oferecer um serviço que pode ser confiável, enquanto a remoção de falhas e previsão de falhas pretendem chegar a confiança, justificando que o funcional e as especificações de confiabilidade e segurança são adequados e que o sistema está pronto para encontrá-los.

3.3.2 Falhas

A Taxonomia de Falhas

Todas as falhas que podem afetar um sistema durante sua vida são classificadas de acordo com oito pontos de vista básicos, levando às classes de falhas elementares. Se todas as combinações das oito classes de falha elementares foram possível, haveria 256 falhas combinando diferentes classes. No entanto, nem todos os critérios são aplicáveis a todas as classes de falhas. As classes de falhas pertencem a três grandes grupos que se sobrepõem parcialmente:

- Falhas de desenvolvimento, que incluem todas as classes de falhas que ocorre durante o desenvolvimento;
- Defeitos físicos que incluem todas as classes de falha que afetam o hardware;
- Falhas de interação que incluem todas as falhas externas.

O conhecimento de todas as classes de falhas possíveis, permite ao usuário decidir quais classes devem ser incluídas em uma especificação de confiabilidade e segurança.

As Falhas Naturais

Falhas naturais são falhas físicas que são causadas por fenômenos naturais, sem a participação humana. Defeitos de produção são falhas naturais que se originam durante o desenvolvimento. Durante a operação as falhas naturais são ou interna, devido a processos naturais que causam a deterioração física, ou externa, devido a processos naturais que originam fora dos limites do sistema e causar interferência física por penetrar a fronteira *hardware* do sistema (radiação, etc.) ou através da introdução por meio de interfaces de uso (os transientes de energia, as linhas de entrada ruidoso, etc.).

Sem Falhas Feitas por Humanos

A definição de falhas feitas pelo homem (que resultam de ações humanas) incluí a ausência de ações quando as ações devem ser realizadas, ou seja, falhas de omissão, ou simplesmente omissões. Realizando ações erradas leva a falhas de comissão.

As duas classes básicas de falhas feitos pelo homem são distinguidas pelo objetivo do promotor ou dos seres humanos que interagem com o sistema durante o seu uso:

- Falhas maliciosas são introduzidas durante o desenvolvimento do sistema, com o objetivo de causar danos ao sistema durante a sua utilização, ou diretamente durante a utilização;
- Falhas não maliciosas são introduzidas sem objetivos maliciosos.

Vamos considerar primeiramente as falhas não maliciosas. Elas podem ser divididas de acordo com a intenção do desenvolvedor:

- *Nondeliberate*: são falhas devido a erros, isto é, ações involuntárias de que o desenvolvedor, operador, mantenedor, etc. não tem conhecimento;
- Falhas deliberadas: são devido a más decisões, ou seja, ações destinadas que estão erradas e causam falhas.
- *Nonmalicious*: são falhas de desenvolvimento deliberadas resultantes geralmente de *trade-offs*, ou que visa preservar o desempenho aceitável, pelo sistema de facilitação utilização, ou induzida por considerações econômicas. Deliberadas, falhas de interação *nonmalicious* pode resultar da ação de um operador ou destinada a superar uma situação imprevista, ou deliberadamente violar um procedimento operacional sem ter percebido as consequências possivelmente prejudiciais desta ação. Falhas deliberadas, *nonmalicious* são muitas vezes reconhecidos como falhas só depois de um comportamento inaceitável do sistema; assim, um fracasso se seguiu. O desenvolvedor (s), operador (es) não percebe no momento em que a consequência de sua decisão foi uma falha.

É geralmente considerado que ambos os erros e más decisões são acidentais, contanto que eles não são feitos com objetivos maliciosos. No entanto, nem todos os erros e mas decisões de pessoas *nonmalicious* são acidentes. Alguns erros muito prejudiciais e decisões muito ruins são feitas por pessoas que não têm competência profissional para fazer o trabalho que realizaram. A taxonomia de falhas completa não deve ocultar esta causa de

falhas; portanto, nós introduzimos um novo particionamento de falhas humanas feitos em *nonmalicious* falhas acidentais, e falhas incompetência.

A questão de como reconhecer falhas se torna importante quando um erro ou uma má decisão tem consequências que levam a perdas econômicas, lesões ou perda de vidas humanas. Em tais casos, o julgamento profissional independente por uma comissão de inquérito ou processo judicial em um tribunal de justiça são susceptíveis de ser necessário para decidir se imperícia profissional estava envolvida. Os esforços feitos pelo homem falharam porque uma equipe ou uma organização inteira não possuem a competência organizacional para fazer o trabalho. Um bom exemplo de incompetência organizacional é o fracasso do desenvolvimento do sistema de AAS, que se destinava a substituir os sistemas antigos de controle de tráfego aéreo nos EUA.

Falhas de desenvolvimento Nonmalicious podem existir em *hardware* e em *software*. Em *hardware*, especialmente nos microprocessadores, algumas falhas de desenvolvimento são descobertas após o início da produção. Tais falhas são chamados de "errata"e estão listadas nas atualizações das especificações. O achado de errata normalmente contínua durante toda a vida útil dos transformadores; portanto, novas atualizações das especificações são emitidas periodicamente. Algumas falhas de desenvolvimento são introduzidas porque as ferramentas criadas pelo homem são defeituosas [4].

Falhas Maliciosas

Falhas maliciosas feitas pelo homem são introduzidas com o objetivo malicioso para alterar o funcionamento do sistema durante o uso. Por causa do objetivo, a classificação de acordo com a intenção e capacidade não é aplicável. Os objetivos de tais falhas são:

- Para interromper ou suspender o serviço, fazendo com que o serviço seja recusado;
- Para acessar informações confidenciais;
- Para modificar o sistema de forma inadequada;

Essas falhas são agrupadas em duas classes:

- Falhas lógicas maliciosas que englobam falhas de desenvolvimento, como cavalos de Troia, bem como falhas operacionais tais como vírus, worms, ou zumbis.
- As tentativas de invasão que são falhas externas operacionais. O caráter externo de tentativas de intrusão não exclui a possibilidade de que eles podem ser realizados por operadores de sistema ou administradores que estão excedendo os seus direitos, e tentativas de intrusão pode utilizar meios físicos para causar falhas: flutuação de energia, radiação, escutas telefônicas, aquecimento / arrefecimento, etc.

Capítulo 4

A Biblioteca BFT-SMaRt

O BFT-SMaRt [7] é uma biblioteca *open source*, escrita em Java, que implementa a replicação de máquina de estados (SMR) e a tolerância a falhas bizantinas (BFT), cujas siglas deram origem ao nome da biblioteca. A biblioteca BFT-SMaRt permite, de uma forma simples, a parametrização de tolerância a falhas bizantinas em aplicações baseadas no paradigma cliente-servidor [25]. O BFT-SMaRt foi projetada para apresentar um alto desempenho, mesmo quando algumas de suas réplicas apresentarem problemas ou sofrerem algum tipo de falha.

Um dos principais objetivos da biblioteca BFT-SMaRt é proporcionar ao desenvolvedor da aplicação, focar sua atenção apenas no desenvolvimento da aplicação. Uma vez que, todo o mecanismo de tolerância a falhas bizantinas é de responsabilidade da biblioteca, o desenvolvedor fica livre para focar no desenvolvimento da aplicação.

A biblioteca BFT-SMaRt é a primeira implementação de BFT SMR que permite reconfigurações no conjunto de réplicas e fornece um suporte eficiente e transparente para serviços duráveis [7]. O projeto do BFT-SMaRt tolera por padrão atraso, perda e corrupção de mensagens e comportamento arbitrário de processos. Além disso, a biblioteca ainda fornece um sistema de assinaturas criptográficas, visando a tolerância a falhas bizantinas e um protocolo SMR para tolerância apenas a falhas por *crash* [7]

4.1 Características da Biblioteca

O BFT-SMaRt foi projetada para tolerar falhas bizantinas e falhas por *crash*. Contudo, para que tal protocolo seja de fato eficiente, o número de réplicas maliciosas deve ser limitado. O BFT-SMaRt assume o modelo de sistema usual para BFT e SMR [8]: $n = 3f + 1$, onde n é o número de servidores (réplicas) no sistema e f o número máximo de máquinas maliciosas. Além disso, o sistema também pode ser configurado para usar somente $n = 2f + 1$ réplicas e tolerar até f falhas por *crash*. Essa parametrização é definida em um arquivo de configuração da própria biblioteca.

Um sistema pode ser estático ou dinâmico [29]. Em um sistema estático, o conjunto de processos permanece inalterado ao longo de todo o seu tempo de vida [29]. Já em um sistema dinâmico, vários outros processos podem ser introduzidos ou retirados do sistema [29]. Em sistemas reais dinâmicos, que utilizam a biblioteca BFT-SMaRt, uma réplica pode falhar por diversos motivos. Contudo, é normal que essa réplica se recupere e que volte a operar no sistema. O problema é que durante o tempo de falha da réplica até

a sua devida volta ao sistema, o estado do sistema pode ter sofrido diversas modificações. Para contornar esse problema, o BFT-SMaRt oferece um protocolo de transferência de estado, permitindo assim que a réplica recupere o estado atual, solicitando o estado atual às outras réplicas do sistema [7]. A esse protocolo foi dado o nome de *Trusted Third Party* (TTP), para gerenciamento do estado das réplicas [11].

4.2 Utilização da API

Primeiramente, para utilização da biblioteca BFT-SMaRt, será necessário baixar a biblioteca do repositório [9] e referenciar a biblioteca ao projeto. Dentro da biblioteca encontra-se alguns arquivos de configurações, dentre eles temos os seguintes arquivos: *system.config* e *hosts.config*.

O arquivo *system.config* é responsável pelas principais configurações da biblioteca [7]. Nele definimos se a aplicação será tolerante apenas a falhas bizantinas, falhas por *crash* ou ambas, o número máximo de servidores (réplicas) no sistema, entre outras configurações. Já o arquivo *hosts.config* é responsável pelo mapeamento dos ID's dos servidores [7]. Nele devemos informar o IP [17] de um servidor e referenciá-lo a um ID específico. Dessa forma, apenas os IP's definidos nesse arquivo, poderão se tornar um servidor aceito no sistema [7].

Além dos arquivos *system.config* e *hosts.config*, dentro da biblioteca terá uma pasta chamada *keys*. A biblioteca BFT-SMaRt utiliza essas chaves para conceder o acesso ao sistema [7]. Cada ID de um componente do sistema, deverá está associado a uma chave válida do sistema, caso contrário, o acesso ao sistema não será permitido.

A biblioteca BFT-SMaRt, nos oferece uma série de classes para uso. Contudo, duas classes da biblioteca BFT-SMaRt, devem ser utilizadas para a criação de uma nova aplicação. As classes *ServiceReplica* e *ServiceProxy*, são as principais classes da biblioteca BFT-SMaRt para a criação de uma nova aplicação.

4.2.1 Classe ServiceReplica

A classe *ServiceReplica* é utilizada no lado do servidor para instanciar um objeto de réplica da aplicação [7]. Esse objeto será utilizado para a comunicação com o BFT-SMaRt, a fim de obter uma nova requisição ou enviar uma resposta ao cliente. Para instanciar esse objeto é necessário passar como argumento, o identificador do servidor (ID). O identificador do servidor será utilizado no arquivo *hosts.config*, para mapeamento de um IP válido no sistema [7].

Após instanciar o objeto *ServiceReplica*, será necessário implementar as seguintes *interfaces* [19]:

- *Executable*: interface responsável por definir os métodos acessados pelo BFT-SMaRt, referentes as requisições do cliente. As requisições do cliente podem ser entregue a aplicação de forma ordenada e não ordenada, uma a uma ou em lote (*batch*). Fica a cargo da aplicação do servidor, implementar a lógica de negócio dessas requisições;
- *Recoverable*: interface responsável por definir os métodos acessados pelo BFT-SMaRt, para gerenciamento do estado da aplicação. O armazenamento do estado

da aplicação é fundamental no caso de algum dos servidores sofrer algum tipo de interrupção. Ao voltar a operar no sistema, o servidor deverá receber o estado atual correto da aplicação de outros servidores do sistema.

Uma maneira de implementar uma nova aplicação no lado do servidor é estender a classe abstrata *DefaultSingleRecoverable*. A classe *DefaultSingleRecoverable* já implementa as interfaces *Executable* e *Recoverable* [7]. Quando estendemos a classe *DefaultSingleRecoverable*, a implementação dos seguintes métodos é obrigatória:

```
public byte[] executeOrdered(byte[] cmd, MsgContext ctx);
public byte[] executeUnordered(byte[] cmd, MsgContext ctx);
public byte[] getSnapshot();
public void installSnapshot(byte[] state);
```

Os dois primeiros métodos (*executeOrdered* e *executeUnordered*) são referentes a interface *Executable*. Ambos os métodos são invocados para a entrega pelo BFT-SMaRt das requisições dos clientes, para a aplicação do servidor [7]. O primeiro método é executado quando o cliente realiza uma requisição ordenada (por exemplo, a escrita de um novo arquivo ou a remoção de um arquivo) e o segundo quando o cliente realiza uma requisição não ordenada (normalmente a leitura de um arquivo). Em ambos os métodos os seguintes argumentos são recebidos:

- byte[] cmd: representa a requisição do cliente, de forma serializada [20]. Como o BFT-SMaRt trafega apenas bytes [7], devemos receber essa requisição também em forma de byte;
- MsgContext ctx: contém os metadados do comando. Por exemplo, a identificação do cliente, horário da requisição, entre outros;

Os dois últimos métodos (*getSnapshot*, *installSnapshot*) são referentes a interface *Recoverable*. Ambos os métodos são responsáveis pelo gerenciamento do estado da aplicação [7]. O método *getSnapshot* é acionado para transferir o estado atual da aplicação, para um novo servidor integrado ao sistema ou para um servidor que havia falhado ou se desconectado por algum motivo e se conectou novamente ao sistema. O método *installSnapshot* é acionado em intervalos fixos (onde o intervalo é parametrizado no arquivo de configuração do BFT-SMaRt [7]), para gerar um novo *snapshot* do estado da aplicação.

4.2.2 Classe ServiceProxy

A classe *ServiceProxy* é utilizada no lado do cliente, para acessar a aplicação do sistema [7]. Cada instância do *ServiceProxy* representa um cliente da aplicação. Utilizamos o objeto instanciado da classe *ServiceProxy* para se comunicar com o BFT-SMaRt, a fim de enviar uma nova requisição ou receber uma resposta dos servidores do sistema. Para instanciar esse objeto é necessário passar como argumento, o identificador do cliente (ID). Nesse caso, o identificador do cliente não é utilizado para mapeamentos no arquivo *hosts.config*. O ID nesse caso, será utilizado para o gerenciamento de clientes, feito pela própria biblioteca BFT-SMaRt [7].

A classe *ServiceProxy* fornece os seguintes métodos, para envio das requisições à aplicação:

```
public byte[] invokeOrdered(byte[] request);
public byte[] invokeUnordered(byte[] request);
public void invokeAsynchronous(byte[] request, ReplyListener listener, int[] receivers, MsgType type);
```

O BFT-SMaRt trafega apenas bytes, em todos esses métodos requisições e respostas devem ser serializadas em bytes [7]. Como trafegamos apenas bytes, os dados por sua vez se tornam genéricos, sendo suportados por qualquer aplicação [6].

Os dois primeiros métodos (*invokeOrdered* e *invokeUnordered*) são invocados para o envio de requisições pelo BFT-SMaRt aos servidores. O primeiro método é executado quando o cliente deseja realizar uma requisição ordenada e o segundo quando o cliente deseja realizar uma requisição não ordenada. Em ambos os métodos, o argumento a ser passado é um *array* de bytes com os dados da requisição (byte[] *request*), devendo este ser serializado antes do envio [7].

Nos dois primeiros métodos, em ambos os casos, o envio é feito de forma síncrona, i.e., a aplicação fica bloqueada esperando uma resposta do servidor [25]. Utilizamos o método *invokeAsynchronous*, quando desejamos enviar uma requisição (tanto ordenada, quanto não ordenada) de forma assíncrona, i.e., sem a necessidade da espera blocante de uma resposta do servidor [25]. Esse método é bem útil para a aplicação continuar a sua execução, enquanto o BFT-SMaRt coleta as respostas dos servidores. O método *invokeAsynchronous* recebe os seguintes argumentos:

- byte[] request: *array* de bytes serializado com os dados da requisição;
- ReplyListener: classe de *callback*, para o gerenciamento das respostas assíncronas;
- receivers: parâmetro para informar quantos clientes devem receber uma resposta;
- MsgType: objeto que define qual será o tipo da requisição (ordenada ou não ordenada).

Capítulo 5

Proposta de Sistema de Arquivos Distribuído

O quinto capítulo descreve a nossa proposta de um Sistema de Arquivos Distribuído. Como pré-requisito para construção do sistema, utilizamos a biblioteca BFT-SMaRt, assumindo como preceito que o nosso sistema possuíra os requisitos básicos de segurança e eficiência [13].

Como descrito no Capítulo 2, um dos preceitos fundamentais de um SAD é que ele forneça ao usuário a ilusão que os dados compartilhados estejam armazenados em um único local [13]. A forma mais simples de se implementar um SAD, ocorre quando existe um parque de servidores responsáveis por tratar as requisições do usuários e armazenar os dados no próprio parque de computadores.

O já citado FARSITE [1] adota um modelo de tratamento e armazenamento centralizados, i.e., todas as requisições dos clientes já são tratadas e armazenadas no próprio parque de recebimento das requisições. Mas centralizar tudo em um mesmo parque de computadores, pode trazer problemas principalmente de desempenho. No caso de uma solicitação de armazenamento de um arquivo relativamente grande, o *directory group* (servidores de tratamento e armazenamento do FARSITE) ficaria sobrecarregado e demandaria bastante tempo para o armazenamento dos dados enviados pelo cliente, dificultando assim o atendimento e tratamento das requisições de outros clientes.

Uma solução alternativa, seria criar dois parques de computadores para o sistema. O primeiro parque seria responsável pelo tratamento das requisições do cliente e o segundo parque seria responsável exclusivamente pelo armazenamento dos dados. Uma atenção com relação a segurança do sistema pode aumentar, porém a sobrecarrega sobre os servidores responsáveis pelo tratamento e recebimento das requisições dos clientes seria bem menor.

Uma aplicação SAD baseada na segunda alternativa de implementação, possuí alguns pontos fundamentais:

- Como enviar as requisições ao parque de computadores responsáveis pelo recebimento e tratamento das informações, de forma que os dados não sofram nenhum tipo de falha, incluindo as bizantinas e as por *crash*;

- Como manter as informações do segundo parque de dados de forma íntegra, i.e., como o primeiro parque de computadores, saberá as informações a cerca dos locais de armazenamento dos dados;
- Qual o melhor método de envio dos dados, a fim de evitar desperdício de tempo e diminuir custos do sistema.

Desta forma, a ideia principal da nossa proposta é construir um SAD baseado na segunda alternativa apresentada, resolvendo os pontos de atenção levantados.

5.1 Modelo do Sistema

O modelo do nosso sistema, se baseia inteiramente na segunda proposta levantada no início do capítulo. Vários sistemas de arquivos se baseiam nessa proposta de desenvolvimento, um dos mais conhecidos na área acadêmica é o *Google File System* [14].

Vamos adotar como uma das premissas, que um número ilimitado de clientes poderá acessar o sistema, desde que esses mesmos clientes possuam uma chave específica (ID) para o acesso.

Através de um canal seguro, os clientes enviaram as suas requisições ao sistema. O sistema então verificará se a requisição do cliente poderá ser atendida, tratará a requisição do cliente e enviará uma resposta ao cliente. De posse dessa resposta, o cliente poderá ou não armazenar os seus dados no parque de armazenamento de dados.

Quando for executada uma requisição pelo cliente, a finalidade principal do sistema é atender essa requisição do cliente de forma ágil e eficiente, em um ambiente seguro e íntegro.

5.2 Arquitetura do Sistema Proposto

Como forma de atender o modelo proposto para o sistema, a seguinte metodologia foi adotada. Caso um cliente deseje realizar uma determinada operação sobre um arquivo no sistema, apenas os metadados [25] do arquivo serão enviados ao primeiro parque de servidores. De posse desses metadados, dependendo da operação requisitada pelo cliente, o primeiro parque de servidores verificará quais são os servidores de armazenamento necessários para o atendimento da requisição. No caso da leitura de um arquivo, em quais servidores de armazenamento uma cópia do arquivo está salva. Já no caso da escrita de um novo arquivo, quais serão os servidores de armazenamento escolhidos para armazenar uma cópia do arquivo. Como forma de facilitação de entendimento do projeto, vamos chamar o primeiro parque de servidores de *Servidores de Metadados* e o segundo parque de servidores de armazenamento de *Storages* [25].

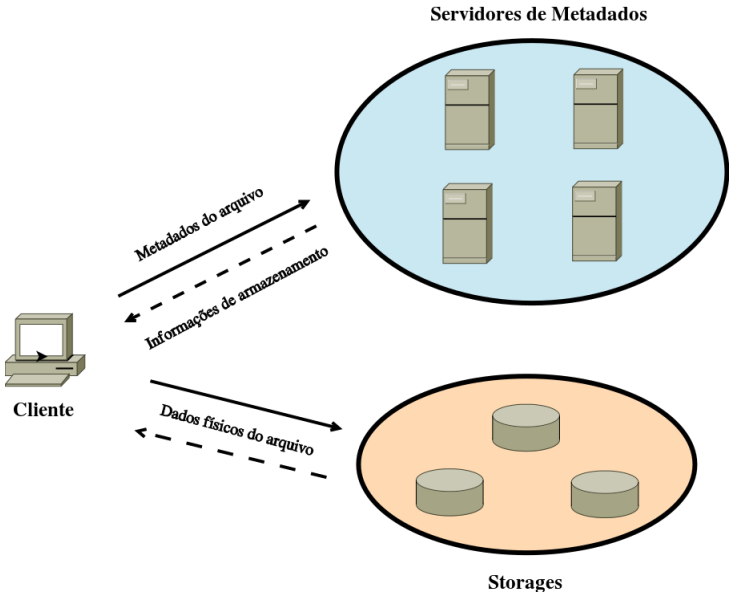

Figura 5.1: Representação da arquitetura do sistema.

Para atender a metodologia proposta, três aplicações foram desenvolvidas, cada uma com uma determinada funcionalidade:

- Aplicação do Cliente: aplicação executada em cada cliente. A principal responsabilidade dessa aplicação é criar uma interface de uso para o cliente. Possuí também como funcionalidades, receber as requisições de um cliente e realizar a comunicação com os Servidores de Metadados e os *Storages*;
- Aplicação do Servidor de Metadados: aplicação executada em cada Servidor de Metadados. A principal responsabilidade dessa aplicação é o recebimento e tratamento das requisições feitas pelos clientes. Possuí também como funcionalidades, o gerenciamento do estado atual da aplicação e o gerenciamento de todos os *Storages*. O gerenciamento dos metadados dos arquivos, será implementado através de SMR usando o BFT-SMaRt;
- Aplicação do *Storage*: aplicação executada em cada *Storage*. As principais responsabilidades dessa aplicação são o recebimento dos dados físicos do cliente e o envio dos dados físicos ao cliente.

Como pode ser visto na Figura 5.2, a ideia é que toda comunicação que envolver a troca de informação com os Servidores de Metadados, seja feita de forma a evitar falhas bizantinas ou falhas por *crash*. Dessa forma, toda comunicação com os Servidores de Metadados, será feita utilizando a biblioteca BFT-SMaRt. Já o envio dos dados físicos do cliente aos *Storages*, ou o recebimento dos dados físicos dos *Storages* ao cliente, será feito via *Sockets TCP/IP* [17].

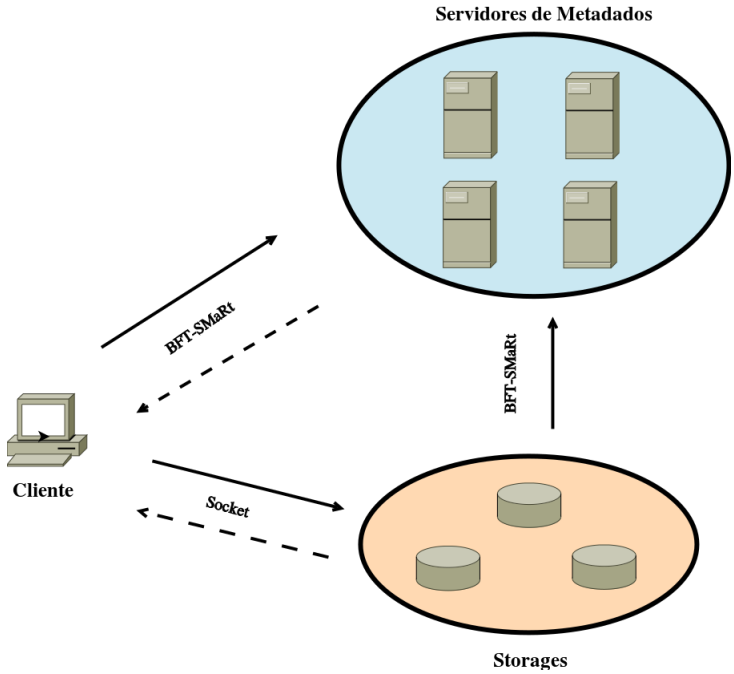

Figura 5.2: Representação da arquitetura do sistema com a biblioteca BFT-SMaRt.

5.2.1 Abordagens de Escrita e Leitura de um Arquivo

Como descrito no Capítulo 4, para uma aplicação realizar a escrita de um determinado arquivo, utilizando a biblioteca BFT-SMaRt e a aplicação seja tolerante a f falhas maliciosas, o número de *Storages* para o armazenamento dos dados do arquivo deve ser: $n = 2f + 1$ [7], onde n é o número de *Storages*. Logo, precisamos de no mínimo três *Storages*, para o sistema conseguir suportar uma única falha maliciosa.

Seguindo essa primeira abordagem, para a escrita de um novo arquivo, primeiramente o cliente enviará uma nova requisição ao sistema, contendo os metadados do arquivo e o número de falhas que o sistema tolera. Os Servidores de Metadados trataram essa nova requisição e então encaminharam uma lista como resposta ao cliente. Essa lista conterá $2f + 1$ melhores *Storages* para armazenamento dos dados do arquivo. Esses melhores *Storages* serão definidos pelos *Storages*, cuja capacidade de armazenamento seja superior ou igual ao tamanho do novo arquivo. O cliente então encaminhará uma cópia do arquivo para cada *Storage* da lista recebida.

Como visito na Figura 5.3, para realizar a leitura do arquivo, o cliente solícita aos Servidores de Metadados o local de armazenamento do arquivo. Os Servidores de Metadados tratam a requisição do cliente e então devolvem ao cliente, uma lista contendo os *Storages* onde o arquivo desejado está armazenado. O cliente então solícita o arquivo, a cada *Storage* da lista, então cada *Storage* envia uma cópia do arquivo para o cliente. De posse de todas as cópias solicitadas do arquivo, o cliente compara todas as cópias recebidas, para verificar a integridade dos dados do arquivo. Ao fim das comparações, caso seja verificado que não há $f + 1$ cópias iguais, todas as cópias são descartadas, pois ocorreu algum problema de integridade no arquivo. Caso contrário, mantemos na pasta do cliente apenas uma cópia do arquivo pertencente a $f + 1$ cópias iguais e descartamos o restante.

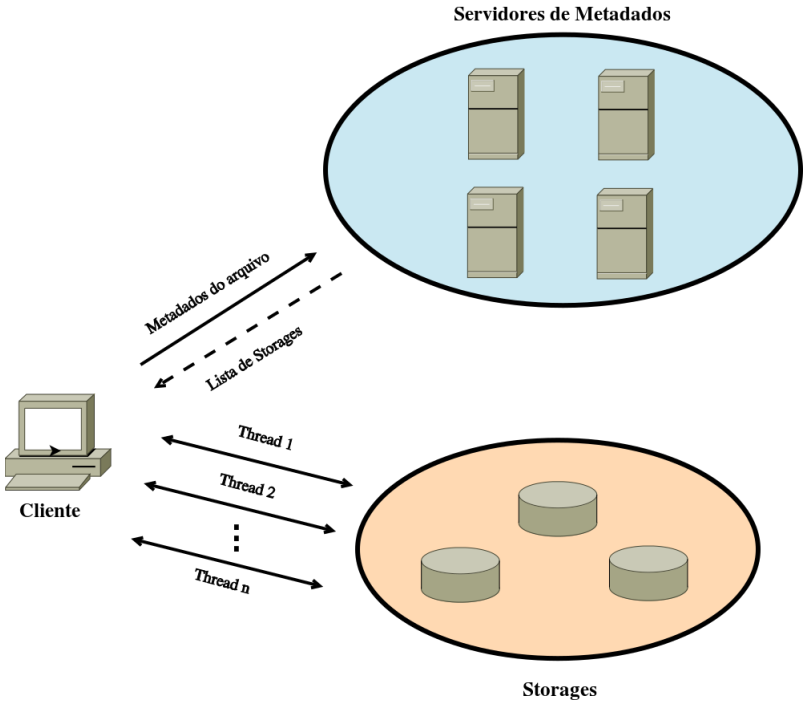

Figura 5.3: Representação da requisição da leitura de um arquivo.

Essa abordagem possuí alguns pontos negativos, entre eles destacamos:

- Maior elevação no tempo de processamento na Aplicação do Cliente, principalmente na etapa de verificação de integridade do arquivo;
- Uma vez que todos os *Storages* devem enviar uma cópia do seu arquivo ao cliente, maior será o uso da rede, para o recebimento dos dados físicos do arquivo de cada *Storage*;
- Um arquivo pode ser modificado $f + 1$ vezes de forma idêntica em f + 1 *Storages*. Na etapa de verificação de integridade do arquivo, será verificado que há $f + 1$ cópias do arquivo iguais, levando a crê que o arquivo contínua íntegro. Logo, falhas de integridade dos dados podem ocorrer, no recebimento dos dados físicos do arquivo.

Como forma de contornar os problemas mencionados, uma nova abordagem foi implementada e utilizada nesse projeto. Como alternativa, utilizamos conceitos de *funções hash* [23], durante o processo de escrita e leitura do arquivo. Nessa nova abordagem, no momento da requisição de escrita de um novo arquivo, antes do envio dos metadados aos Servidores de Metadados, um código *hash* será gerado a partir dos dados físicos do arquivo solicitado. Esse código *hash* será enviado com os demais metadados, aos Servidores de Metadados. Os Servidores de Metadados por sua vez, armazenaram esse código *hash* com os demais metadados, na sua estrutura de diretórios e arquivos. Semelhante a abordagem anterior, os Servidores de Metadados encaminharam a aplicação do cliente, uma lista com os melhores *Storages* e o cliente encaminhará os dados físicos do arquivo ao *Storages* dessa lista.

Para realizar a leitura do arquivo, o cliente solicita aos Servidores de Metadados, o local de armazenamento do arquivo. Os Servidores de Metadados tratam a requisição do

cliente e então devolvem ao cliente, uma lista contendo os *Storages* onde o arquivo desejado está armazenado e o código *hash* (*C1*) do arquivo solicitado. O cliente então solícita o arquivo, apenas ao primeiro *Storage* da lista. Somente o primeiro *Storage* da lista, envia uma cópia do arquivo ao cliente. De posse da cópia desse arquivo, o cliente gera um novo código *hash* desse arquivo (*C2*). Para verificar a integridade do arquivo, comparamos os códigos *hash*'s. Caso $C1 = C2$, o arquivo recebido está íntegro. Caso contrário, uma nova solicitação é realizada ao próximo *Storage* da lista recebida. O processo será encerrado, caso haja alguma cópia do arquivo íntegra ou se chegue ao fim da lista de *Storages*.

A grande vantagem dessa nova abordagem é que não precisará de $2f + 1$ *Storages*, para armazenamento dos arquivos. Uma vez que, caso tenhamos definido cem *Storages* para armazenamento do arquivo. Mesmo que noventa e nove *Storages* venham a sofrer algum tipo de falha no arquivo. Para o cliente receber o arquivo de forma íntegra, precisamos apenas que o arquivo em um único *Storage*, não sofra nenhum tipo de falha. Logo, adotamos a seguinte fórmula, para definição do número de *Storages* de armazenamento: $n = f + 1$, onde n é o número de *Storages* e f o número de falhas aceitáveis.

Para fins didáticos, as duas abordagens de escrita e leitura foram implementadas no sistema. Mais adiante, demonstraremos a implementação em detalhes das operações de escrita e leitura, os resultados de uso e as comparações com a utilização de cada abordagem no sistema.

5.3 Metodologia

O caminho trilhado para desenvolver esse sistema, se baseou em conceitos de Engenharia de Software. Os conceitos utilizados se originam da programação orientada a objetos, mas especificamente utilizando a linguagem Java [21], como primitiva para o desenvolvimento. O padrão em camadas [24] foi o escolhido para a criação e gerenciamento do projeto, de forma a modularizar todas as classes envolvidas no sistema.

5.3.1 Implementação

Todo o projeto foi escrito utilizando a linguagem Java e o padrão JavaDoc [16]. Com isso, manutenções futuras e novas integrações no sistema, serão feitas de forma mais simples e eficiente.

Para implementação do sistema, todo o código fonte do projeto foi dividido em sete pacotes. Cada pacote é composto por uma ou várias classes, responsáveis por um ou mais componentes do sistema. Os pacotes são:

- Cliente: pacote responsável pelas classes referentes aos dados do cliente;
- Diretorio: pacote responsável pelas classes referentes aos dados de um diretório;
- Servidor: pacote responsável pelas classes referentes aos dados de um Servidor de Metadados;
- *Storage*: pacote responsável pelas classes referentes aos dados de um *Storage*;
- Interfaces: pacote responsável pelas interfaces [19] do sistema;
- Testes: pacote responsável pelas classes referentes aos testes no sistema;

- Utils: pacote com classes utilitárias para uso no sistema.

Material Utilizado

- O sistema operacional utilizado foi o Ubuntu 16.04 LTS;
- Os notebooks utilizados foram um Sony Vaio e um DELL;
- O ambiente de desenvolvimento foi o Eclipse Luna;
- O ambiente para realização dos testes foi o Emulab [30];
- A ferramenta para geração e exportação dos gráficos foi o Gnuplot 5.0.

Os códigos-fonte estão disponíveis para *download* nos seguintes repositórios:

- https://github.com/guilherme110/projetoFinal.git
- https://guilherme110.github.io/projetoFinal/

5.3.2 Aplicação do Cliente

A aplicação do cliente foi construída para servir como interface ao cliente. Como mencionado no Capítulo 4, para implementarmos uma aplicação do cliente, que utilize a biblioteca BFT-SMaRt, precisamos utilizar a classe *ServiceProxy*, para comunicação com os Servidores de Metadados. De início, para utilização da classe *ServiceProxy*, devemos passar como parâmetro o identificador do cliente (ID). Esse parâmetro será informado na forma de argumento na inicialização da aplicação. Os seguintes argumentos, devem ser informados na inicialização da aplicação:

- O identificador único do cliente. Esse parâmetro será utilizado no arquivo de configurações da biblioteca BFT-SMaRt, para mapeamento da aplicação [7];
- O número de falhas que o sistema poderá suportar;
- O diretório local do usuário.

Após as validações necessárias nos argumentos informados pelo cliente, será necessário instanciar uma série de novos objetos, para uso frequente na aplicação. Primeiramente instanciamos um objeto do tipo *Cliente*, para representação dos dados do cliente. Nesse momento também é instanciado um novo objeto do tipo *ServiceProxy*, para realizar a comunicação com os Servidores de Metadados. Em seguida, o diretório *home* é definido como o diretório raiz do cliente. Por último é instanciado um objeto do tipo *Cliente-Servico*, para gerenciamento dos serviços disponíveis do cliente. Ao decorrer do projeto, todos esses objetos de uso frequente na aplicação, serão detalhados.

```
public static void criaCliente(String idCliente, int fNumeroFalhas,
  String localArmazenamento) {
  cliente = new Cliente();

  cliente.setIdCliente(Integer.parseInt(idCliente));
  cliente.setDiretorioClienteAtual(new ArrayList<String>());
  cliente.getDiretorioClienteAtual().add("home");
  cliente.setfNumeroFalhas(fNumeroFalhas);
  cliente.setLocalArmazenamento(localArmazenamento);
```

```
  try {
      KVProxy = new ServiceProxy(cliente.getIdCliente());
      clienteServico = new ClienteServico(KVProxy);
  } catch (Exception e) {
      System.out.println("Erro de comunicação com os servidores!");
      System.exit(-1);
  }
}
```

Caso a comunicação com a biblioteca BFT-SMaRt seja estabelecida com sucesso, a aplicação já estará apta para receber novas requisições do cliente. O objeto *KVProxy* será utilizado durante toda aplicação, para a comunicação com o BFT-SMaRt.

Como forma de facilitar a navegação do cliente na aplicação, para a realização de novas requisições, foi criado um menu com as opções disponíveis da aplicação. As seguintes opções estão disponíveis para o cliente:

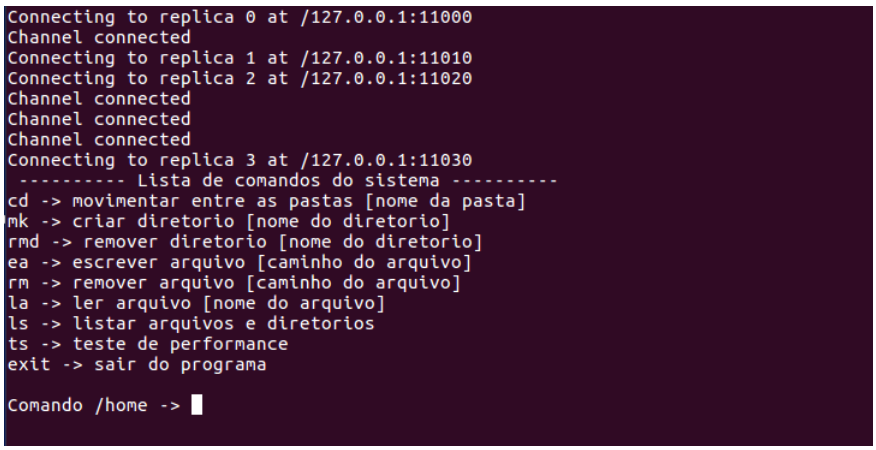

Figura 5.4: Representação do menu de opções da Aplicação do Cliente.

- cd: opção para movimentar entre as pastas do sistema. Recebe como argumento, o nome da pasta de destino da movimentação. O argumento ".." é informado para acessar o diretório diretamente superior do diretório atual;
- mk: opção para criar um diretório. Recebe como argumento, o nome do novo diretório. Não é possível criar dois diretórios como o mesmo nome em um mesmo local;
- rmd: opção para remover um diretório. Recebe como argumento, o nome do diretório a ser removido;
- ea: opção para escrever um novo arquivo no sistema. Recebe como argumento, o caminho onde se encontra o arquivo. Não há restrição em relação a extensão do arquivo e ao tamanho do arquivo. Não é possível salvar em um mesmo diretório, arquivos com o mesmo nome e extensão;
- rm: opção para remover um arquivo. Recebe como argumento, o nome do arquivo a ser removido;
- la: opção para ler um arquivo do sistema. Recebe como argumento o nome do arquivo a ser lido;

- ls: opção para listar os arquivos e pastas do diretório atual;
- ts: opção para realizar testes de performance no sistema;
- exit: opção para sair da aplicação.

Como forma de facilitar a navegação de novos usuários ao sistema, a nomenclatura dos comandos foi aproximada a nomenclatura de comandos existentes no sistema UNIX [5]. Outra característica é que o sistema é *case-sensitive* [5], i.e., um comando "cd" é diferente de um comando "CD".

Camadas da Aplicação

Para cada requisição efetuada pelo cliente, um serviço é realizado. Para gerenciar os serviços disponíveis na Aplicação do Cliente, uma nova camada foi criada, a essa camada demos o nome de *ClienteServico*. Essa camada tem como responsabilidade:

- Gerenciar os serviços da Aplicação do Cliente;
- Verificar se os requisitos de uma determinada requisição estão sendo atendidos;

Como mencionado no Capítulo 4, a classe *ServiceProxy* nos disponibiliza três métodos para o envio das requisições [7]. Nesse projeto apenas os dois primeiros métodos foram utilizados, i.e., apenas os métodos *invokeOrdered* e *invokeUnordered*, logo o método *invokeAsynchronous* não foi utilizado no desenvolvimento do projeto. Para determinar qual método será utilizado na comunicação com o BFT-SMaRt, uma nova camada foi implementada, para realizar diretamente a comunicação com a biblioteca BFT-SMaRt. Essa camada foi chamada *MapDiretorio*.

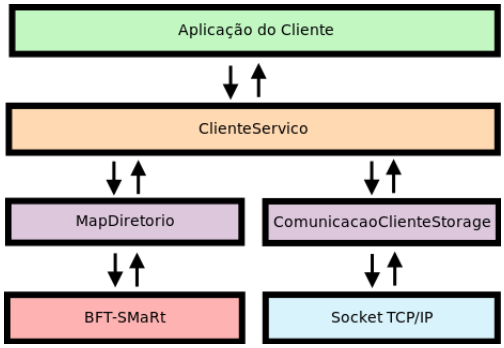

Figura 5.5: Representação das camadas da Aplicação do Cliente.

Como visto na Figura 5.5, toda requisição aos Servidores de Metadados, passa pela camada *MapDiretorio*, cuja as responsabilidades são:

- Verificar se a requisição do cliente deve ser ordenada ou não ordenada, i.e., determinar qual método de comunicação será executado (invokedOrdered ou invokedUnordered);
- Serializar os dados da requisição em bytes, para serem enviados ao BFT-SMaRt;

- Receber e serializar a resposta da requisição feita ao BFT-SMaRt.

Um outro ponto importante na Aplicação do Cliente é a forma que se dará a comunicação com os *Storages*. Para isso, uma nova camada foi criada, essa camada foi chamada de *ComunicacaoClienteStorage*. Essa camada tem como responsabilidade, os seguintes pontos:

- Enviar os dados da aplicação do cliente para os *Storages*;
- Receber os dados dos *Storages* para a aplicação do cliente.

5.3.3 Aplicação do Servidor de Metadados

A Aplicação do Servidor de Metadados foi construída para servir como recepção, tratamento e resposta as requisições do cliente. Como mencionado no Capítulo 4, para implementarmos uma aplicação do servidor (na biblioteca BFT-SMaRt chamado de réplica [7]) utilizando a biblioteca BFT-SMaRt, uma das formas é estender a classe abstrata *DefaultSingleRecoverable*. Após a extensão da classe *DefaultSingleRecoverable*, a implementação dos seguintes métodos é obrigatória: *executeOrdered, executeUnordered, getSnapshot* e *installSnapshot*.

Além da implementação dos métodos obrigatórios da classe *DefaultSingleRecoverable*, precisamos instanciar um objeto do tipo *ServiceReplica*. De início, para utilização da classe *ServiceReplica*, devemos passar como parâmetro, o identificador do servidor (ID). Definimos que o parâmetro seria informado em forma de argumento, na inicialização da aplicação. Os seguintes argumentos foram definidos, para serem informados na inicialização da aplicação:

- O identificador único do servidor. Utilizado no arquivo de configurações da biblioteca BFT-SMaRt, para mapeamento da aplicação;
- Uma *flag* (true/false), indicando se aquele servidor está habilitado para realização de testes;
- O número de intervalos para a realização dos testes.

Os dois últimos parâmetros serão utilizados apenas nos casos de testes do sistema, eles serão explicados mais adiante. Caso todos os argumentos informados na inicialização da aplicação sejam válidos, um objeto de serviço da aplicação é criado. Nesse momento é instanciado um objeto do tipo *ServiceReplica*, para realizar a comunicação com o BFT-SMaRt e instanciado os objetos *arvoreDiretorio, tabelaStorage* e *servidorService*. Apenas a camada de serviços [24] foi implementada na aplicação do Servidor de Metadados, para o devido tratamento das requisições dos clientes.

```
public ServidorMetaDados(int idServidor, boolean mensurarTestes, int  numeroIntervalo) {
    arvoreDiretorio = new ArvoreDiretorio();
    servidorServico = new ServidorServico();
    tabelaStorage =  new HashMap<Integer,Storage>();

    new ServiceReplica(idServidor, this, this);
}
```

Árvore de Diretórios e Tabela de Storages

Em Estrutura de Dados temos o conceito de árvore [27]. Cada árvore possui ramos e folhas (ou nó) [27]. Utilizamos a estrutura de árvores, como forma de gerenciar o estado da nossa aplicação. A princípio nossa árvore é composta pelos seguintes objetos:

- Arquivo: objeto para representar um arquivo salvo no sistema. Possuí os seguintes atributos:
  - Nome do arquivo: identificador do arquivo salvo;
  - Tamanho do arquivo: tamanho do arquivo salvo;
  - Lista de ID's dos *Storages*: lista que contém o ID de cada *Storage* que armazena uma cópia do arquivo;
  - Código *hash*: código gerado a partir dos dados físicos do arquivo.
- Diretorio: objeto para armazenar um ou mais arquivos ou subdiretórios. Possuí os seguintes atributos:
  - Nome do diretório: identificador do diretório criado;
  - Lista de dados: lista com todos os arquivos e subdiretórios salvos no diretório;

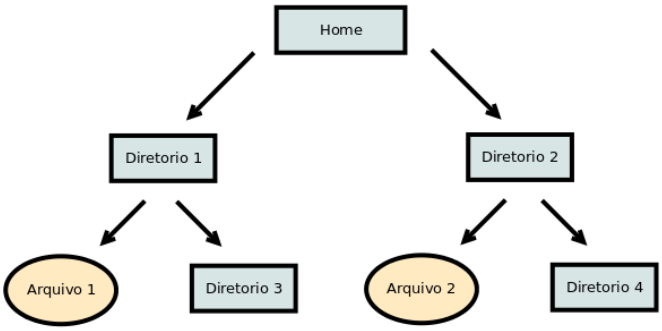

Figura 5.6: Representação da Árvore de Diretórios.

Como visto na Figura 5.6, cada nó da árvore pode ser um diretório ou um arquivo. No caso do diretório, ele pode possuir um ou mais arquivos ou subdiretórios. O diretório raiz por padrão será o diretório *home*. Toda a estrutura de diretórios do sistema é feita por uma única árvore, compartilhada entre todos os clientes. Logo, caso um cliente escreva um novo arquivo no sistema, todos os outros clientes terão acesso a esse arquivo.

Além do gerenciamento dos diretórios e arquivos do cliente, precisamos gerenciar também o estado dos *Storages* do sistema. Para isso, criamos uma Tabela de *Storages*, para o gerenciamento de todos os *Storages*. Conforme pode ser observado na Tabela 5.1, a cada novo *Storage* integrado ao sistema, o mesmo deve enviar os seus dados para os Servidores

de Metadados. De posse dos dados desse novo *Storage*, uma nova linha é acrescentada na tabela. A cada nova requisição de escrita ou remoção de um arquivo no sistema, a tabela é atualizada.

Nome	ID	Espaço Livre	Endereço	Porta	Lista de Arquivos	Local de Armazenamento
Storage A	7000	100000000	10.1.1.7	10101	arquivo A, arquivo B	/home/StorageA
Storage B	7001	50000000	10.1.1.8	10102	arquivo A, arquivo C	/home/StorageB
Storage C	7002	20000000	10.1.1.9	10103	arquivo B, arquivo C	/home/StorageC
Storage N	7003	200000000	10.1.1.10	10104	arquivo A, arquivo B, arquivo C	/home/StorageN

Tabela 5.1: Representação da Tabela de *Storages*

Como nossa Árvore de Diretórios e a Tabela de *Storages* representam o estado atual da aplicação, esses dados devem ser idênticos e replicados em todos os Servidores de Metadados. Conforme mencionado no Capítulo 4, a biblioteca BFT-SMaRt se responsabiliza por manter a integridade do estado da aplicação, entre todos os servidores [7]. Para isso, implementamos os métodos *getSnapshot* e *installSnapshot*, serializando os objetos [20]: *arvoreDiretorios* e *tabelaStorages*. Com isso, a biblioteca BFT-SMaRt estará pronta para gerenciar o estado da aplicação.

```
@Override
public void installSnapshot(byte[] state) {
    try {
        ByteArrayInputStream bis = new ByteArrayInputStream(state);
        ObjectInput in = new ObjectInputStream(bis);
        arvoreDiretorio = (ArvoreDiretorio) in.readObject();
        tabelaStorage = (Map<Integer, Storage>) in.readObject();
        in.close();
        bis.close();
    } catch (ClassNotFoundException ex) {
        Logger.getLogger(BFTMapServer.class.getName()).log(Level.SEVERE, null, ex);
    } catch (IOException ex) {
        Logger.getLogger(BFTMapServer.class.getName()).log(Level.SEVERE, null, ex);
    }
}


@Override
public byte[] getSnapshot() {
    try {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutput out = new ObjectOutputStream(bos);
        out.writeObject(arvoreDiretorio);
        out.writeObject(tabelaStorage);
        out.flush();
        bos.flush();
        out.close();
        bos.close();
        return bos.toByteArray();
    } catch (IOException ex) {
        Logger.getLogger(ServidorMetaDados.class.getName()).log(Level.SEVERE, null, ex);
        return new byte[0];
    }
}
```

5.3.4 Aplicação do Servidor de Armazenamento

A Aplicação do *Storage* foi construída para o gerenciamento e armazenamento dos dados físicos do cliente. Primeiramente devemos criar uma comunicação com os Servidores

de Metadados, para integração do *Storage* ao sistema. Para isso, utilizamos a biblioteca BFT-SMaRt para realizar a comunicação com os Servidores de Metadados. Semelhante a Aplicação do Cliente, será necessário instanciar um objeto do tipo *ServiceProxy*, passando como parâmetro o ID do *Storage*. Novamente esse ID será informado como argumento, na inicialização da aplicação. Os seguintes argumentos devem ser informados na inicialização da aplicação:

- O identificador único do *Storage*. Utilizado no arquivo de configurações da biblioteca BFT-SMaRt, para mapeamento da aplicação;
- A porta de rede [17] do *Storage*. Utilizado na comunicação com o cliente;
- O tamanho máximo de armazenamento de dados do *Storage*;
- A pasta de armazenamento dos dados dos arquivos.

Caso todos os argumentos informados sejam válidos, instanciamos um novo objeto do tipo *ServerSocket* [22], para gerenciar a comunicação com os clientes. Nesse momento também instanciamos um novo objeto do tipo *Storage*, onde o mesmo possuí os seguintes argumentos:

- Nome do *Storage*: nome do host definido no próprio sistema operacional do servidor [5];
- Endereço lógico do sistema. Nesse caso utilizamos o endereço IP do servidor [17];
- Porta de rede para conexão [17], definida na inicialização da aplicação.
- Espaço livre total de armazenamento, definido na inicialização da aplicação;
- Lista de todos os arquivos armazenados no *Storage*;
- Identificador do *Storage*, definido na inicialização da aplicação.
- Local padrão para armazenamento dos arquivos, definido na inicialização da aplicação.

```
private static boolean iniciaServidor(String[] args) {
    storage = new Storage(Integer.parseInt(args[0]), Integer.parseInt(args[1]),
            Long.parseLong(args[2]), args[3] ,new ArrayList<Arquivo>());

    try {
        serverSocket = new ServerSocket(storage.getPortaConexao());
        storage.setNomeStorage(serverSocket.getInetAddress().getLocalHost().getHostName());
        storage.setEnderecoHost(serverSocket.getInetAddress().getLocalHost().getHostAddress());

        KVProxy = new ServiceProxy(storage.getIdStorage());
        if (!enviaDadosStorage(storage))
            throw new Exception();
        System.out.println("Dados enviados para o servidor de meta dados com sucesso!");
    } catch (Exception e) {
        System.out.println("Erro de comunicação com os servidores!");
        System.exit(-1);
    }
}
```

Em seguida, criamos uma nova comunicação com o BFT-SMaRt, instanciando um novo objeto do tipo *ServiceProxy*. Para manter o estado do sistema íntegro, o novo *Storage* estabelece uma comunicação com os Servidores de Metadados, passando suas informações aos Servidores de Metadados. Para isso, realizamos o método *invokeOrdered* para o envio dos dados de forma ordenada.

Após receber a requisição do *Storage*, os Servidores de Metadados atualizam as suas Tabelas de *Storages*, incluindo uma nova linha na tabela, referente ao novo *Storage*. Ao fim do processo, o servidor permanecerá escutando a sua porta de comunicação, aguardando novas requisições dos clientes.

5.3.5 Operações de Escrita e Leitura no Sistema

Escrita do Arquivo Sem Hash (primeira abordagem)

Como citado no início do capítulo, implementamos duas abordagens para a escrita do arquivo. A primeira abordagem não utiliza o conceito de *hash*, enquanto a segunda abordagem utiliza o conceito de *hash*. Na primeira abordagem, enviamos os seguintes dados aos Servidores de Metadados, utilizando a biblioteca BFT-SMaRt:

- Identificador da operação: identificador utilizado no lado do Servidor de Metadados, para identificar qual operação está sendo requisitado.
- Nome do arquivo: identificador utilizado na busca do arquivo na árvore de diretório. Sua utilização também é necessária, para evitar duplicidade de arquivos com o mesmo nome, em um mesmo diretório;
- Tamanho do arquivo: utilizado para determinar os *Storages* que possuem espaço suficiente para armazenamento do arquivo;
- Diretório atual do cliente: utilizado para armazenar o arquivo no seu respectivo diretório de destino;
- Número de *Storages* para armazenamento: utilizado para determinar o número de *Storages* necessários para o armazenamento do arquivo, de acordo com o que foi definido pelo cliente.

A posição e o tipo de dado de cada informação enviada aos Servidores de Metadados são cruciais, uma vez que utilizaremos a posição de cada informação, para identificar a informação nos Servidores de Metadados.

Ao serializar em bytes os dados da requisição, o próximo passo é invocar o método de envio da requisição da biblioteca BFT-SMaRt. Nesse caso, por se tratar de uma requisição que deve ser do tipo ordenada, utilizamos o método *invokeOrdered* passando como parâmetro os dados serializados da requisição. Após o envio da requisição, a própria biblioteca BFT-SMaRt ficará encarregada de acionar os Servidores de Metadados e acionar o método responsável pelas requisições ordenadas.

No momento em que o Servidor de Metadados receber uma nova requisição, a sua primeira tarefa será a de tratar os dados recebidos da requisição. Algumas validações devem ser realizadas nos dados recebidos, visando evitar duplicidade de arquivos com o mesmo nome, em um mesmo diretório. Após as validações na requisição, um objeto do

tipo *Arquivo* será instanciado a partir dos dados da requisição. Utilizaremos esse objeto, para representar o novo arquivo a ser escrito no sistema.

Em seguida, serão verificados os melhores *Storages* para armazenamento do arquivo. O sistema realiza apenas uma validação, para que o *Storage* seja definido como o melhor local para armazenamento do arquivo. Caso a capacidade de armazenamento do *Storage*, seja superior ou igual ao tamanho do novo arquivo, ele será definido como melhor *Storage*. Nesse momento, adicionamos o ID de cada *Storage* escolhido, no atributo *listaIdStorage* do objeto *Arquivo*. Utilizaremos esse atributo, para consultas posteriores no arquivo.

Após a definição dos melhores *Storage*, devemos atualizar a Tabela de *Storages*. No caso da escrita, a linha que representa o *Storage* é atualizada, de forma que a informação a cerca do tamanho disponível naquele *Storage*, seja atualizada com a adição desse novo arquivo.

Em seguida, devemos atualizar a Árvore de Diretórios com o novo arquivo. Um novo nó na árvore será criado e o objeto *Arquivo* será adicionado. Após a atualização, uma lista contendo os melhores *Storages* para armazenamento do arquivo é encaminhada ao cliente.

Por último, o cliente enviará uma cópia do arquivo, para cada *Storage* da lista recebida. Para a primeira abordagem de escrita, a fim de evitar as falhas maliciosas, o arquivo deverá ser enviado para $2f + 1$ *Storages*. Para não comprometer tanto o desempenho do sistema, utilizamos o conceito de *Thread* [15], para o envio dos dados do arquivo aos *Storages*.

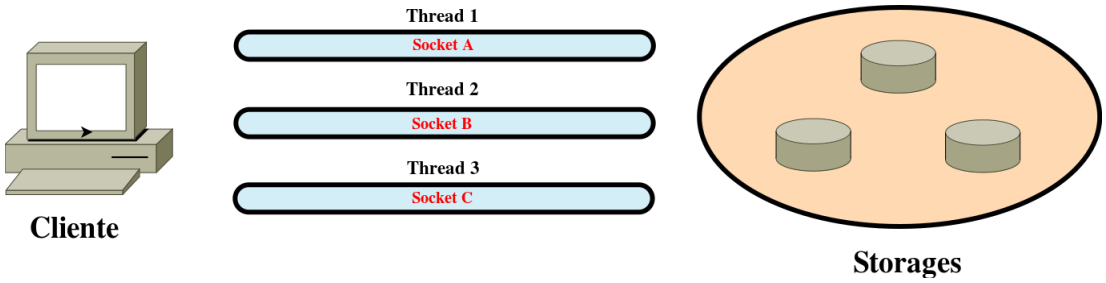

Figura 5.7: Representação da comunicação do Cliente com os *Storages*.

Conforme pode ser visto na Figura 5.7, para cada *Storage* uma *Thread* é criada. Dentro de cada *Thread*, uma comunicação é estabelecida com um *Storage* via *Socket*. Com isso, conseguimos enviar uma cópia do arquivo para cada *Storage*, de forma paralela [15]. Uma atenção com relação a segurança do sistema pode aumentar, principalmente em relação a concorrência dos dados [15], porém o tempo de tratamento e envio de dados aos *Storages* é bem menor. Nesse momento, a camada *ComunicacaoClienteStorage* é responsável por gerenciar cada *Thread* de comunicação. Cada *Thread* será responsável por abrir um *Socket* de comunicação com um *Storage*. Nesse momento, a Aplicação do Cliente ficará bloqueada, aguardando o fim de execução de todas as *Thread's*.

Após a abertura do *Socket*, os seguintes dados são serializados e encaminhados ao *Storage*:

- Identificador da operação: código de identificação da requisição solicitada ao *Storage*;
- Objeto arquivo: objeto utilizado para identificação do arquivo no *Storage*;

- Dados do arquivo: *array* de bytes dos dados do arquivo. Utilizamos a biblioteca Apache IOUtils [2] para envio dos dados, uma vez que a biblioteca já possuí os métodos necessários para o envio de grandes dados.

No momento em que o *Storage* receber uma nova requisição, a sua primeira tarefa será a de tratar os dados recebidos da requisição. Novamente utilizamos o conceito de *Thread's*, para tratar as requisições do cliente. Para cada nova requisição ao *Storage*, uma *Thread* será criada. Nesse momento, uma nova camada chamada *TrataCliente* será responsável pelo gerenciamento das *Thread's*. No caso da escrita de dados, o *Storage* criará e armazenará o arquivo na sua pasta de armazenamento. Em seguida, a *Thread* será encerrada e nenhuma mensagem de resultado será encaminhada ao cliente. Ao fim do processamento de todas as *Thread's* de comunicação com os *Storages*, a aplicação do cliente continuará a sua execução e retornará uma mensagem a tela da Aplicação do Cliente.

Escrita do Arquivo Com Hash (segunda abordagem)

Nessa segunda abordagem, como tentativa de diminuição do tempo de execução da operação de escrita, utilizamos o conceito de *hash*. Primeiramente na Aplicação do Cliente devemos gerar o código *hash* do arquivo, para isso o algoritmo SHA-256 [23] foi utilizado. O algoritmo SHA-256 foi escolhido por possuir como principal característica, processar mensagens com até 2^{64} bits e trabalhar com palavras de 32 bits [23], atendendo o escopo do nosso projeto. A própria linguagem Java nos da suporte para a geração de códigos *hash*, para isso utilizamos a biblioteca *MessageDigest* [21].

```
public static String geraHashArquivo(FileInputStream inputArquivo) {
    MessageDigest algorithm;
    try {
        algorithm = MessageDigest.getInstance(ALGORITMO_HASH);
        byte messageDigest[] = algorithm.digest(IOUtils.toByteArray(inputArquivo));

        StringBuilder hexString = new StringBuilder();
        for (byte b : messageDigest) {
            hexString.append(String.format("%02X", 0xFF & b));
        }
        return hexString.toString();

    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

Utilizamos o método *getInstance* da biblioteca MessageDigest, passando como parâmetro, o algoritmo de geração do código *hash*, no nosso caso o SHA-256. Como saída recebemos um *array* de bytes, representando o código *hash* gerado. Como forma de evitar desperdício de espaço, representamos o *array* de bytes em formato hexadecimal [23].

A partir desse ponto, as etapas para a escrita do arquivo são semelhantes as mencionadas na primeira abordagem, com algumas alterações:

- Será enviado aos Servidores de Metadados, além das informações mencionadas na primeira abordagem, o código *hash* gerado dos dados do arquivo em formato hexadecimal. O armazenamento do código *hash* será útil no momento da leitura do arquivo, vamos descrever isso posteriormente;

- O arquivo será enviado apenas para $f + 1$ *Storages*.

Leitura do Arquivo Sem Hash (primeira abordagem)

Na primeira abordagem de leitura, utilizamos a biblioteca BFT-SMaRt, para enviar os seguintes dados aos Servidores de Metadados:

- Identificador da operação: utilizado no lado do Servidor de Metadados, para identificar qual operação está sendo requisitada;
- Nome do arquivo: utilizado para identificação do arquivo na Árvore de Diretórios;
- Diretório atual do cliente: utilizado para buscar o arquivo no seu respectivo diretório.

Em seguida, serializamos os dados da requisição e invocamos o método para o envio da requisição através da biblioteca BFT-SMaRt. Nesse caso, por se tratar de uma requisição que necessariamente não deve ser do tipo ordenada, utilizamos o método *invokeUnordered*, passando como parâmetro os dados serializados da requisição.

No momento em que o Servidor de Metadados receber a requisição, a sua primeira tarefa será a de tratar os dados recebidos da requisição. Algumas validações devem ser realizadas nos dados recebidos, para verificar se o arquivo de fato se encontra no diretório solicitado. Caso nenhuma invalidade na requisição tenha sido encontrada, instanciamos um objeto do tipo arquivo, utilizando como parâmetro os dados informados pelo cliente (nome do arquivo e diretório atual do cliente), a partir da Árvore de Diretórios. Como mencionado no início do capítulo, o objeto do tipo arquivo possuí um atributo chamado *listaIdStorage*. Esse atributo, possuí os ID's de todos os *Storages* que o arquivo está armazenado. Utilizamos esse atributo, para percorrer a Tabela de *Storages* e instanciar um novo objeto do tipo *Storage*, para cada ID da *listaIdStorage*. Cada novo objeto do tipo *Storage*, será armazenado em uma nova lista, que será encaminhada ao cliente como resposta. Caso qualquer exceção ocorra, uma mensagem de erro é serializada e encaminhada como resposta ao cliente.

De posse da lista que contêm os *Storages* onde o arquivo se encontra armazenado, o próximo passo da Aplicação do Cliente é solicitar esses arquivos aos *Storages*. Para isso, utilizamos um método semelhante ao utilizado nas escritas dos arquivos, o uso de *Thread's* na comunicação com os *Storages*. Nesse contexto, cada *Thread* também será responsável por abrir uma nova comunicação via *Socket TCP/IP* com um *Storage*. Utilizamos a camada *ComunicacaoClienteStorage* para gerenciar a comunicação com o *Storage*. Nesse momento, a Aplicação do Cliente ficará bloqueada aguardando o fim de execução de todas as *Thread's*.

Após a abertura do *Socket*, os seguintes dados são serializados e encaminhados ao *Storage*:

- Identificador da operação: código de identificação da requisição solicitada ao *Storage*;
- Objeto arquivo: objeto utilizado para identificação do arquivo no *Storage*;

No momento em que o *Storage* receber a requisição, a sua primeira tarefa é tratar os dados recebidos pelo cliente, para verificar qual requisição foi solicitada pelo cliente e os

dados necessários para atender a requisição. Semelhante ao mecanismo de escrita, utilizamos a camada *TrataCliente* para gerenciar uma nova *Thread* de tratamento da requisição. O *Storage* verificará se existe o arquivo solicitado em seu local de armazenamento, em caso positivo, enviará o arquivo ao cliente serializado em um *array* de bytes.

A aplicação então criará um arquivo temporário para cada arquivo recebido de um *Storage*. Cada arquivo temporário ficará armazenado na pasta de armazenamento do cliente. De posso de todos os arquivos temporários, será executado um método para verificar a integridade dos dados recebidos. Para verificar a integridade dos dados, comparamos byte a byte cada arquivo recebido. Conforme mencionado no início do capítulo, caso tenhamos $f + 1$ arquivos diferentes, todos os arquivos temporários são removidos e uma mensagem indicando que houve erro de integridade nos dados gerada na aplicação. Caso contrário, um arquivo pertencente a $f + 1$ arquivos iguais é mantido na pasta do cliente e o restante dos arquivos temporários são removidos.

Leitura do Arquivo Com Hash (segunda abordagem)

Nessa segunda abordagem, utilizaremos o conceito de *hash*, a fim de diminuir os custos da operação. Semelhante a primeira abordagem, encaminhamos aos Servidores de Metadados apenas o nome da operação solicitada, o nome do arquivo e o diretório atual do cliente. De posse dessas informações o Servidor de Metadados instanciará um novo objeto do tipo *Arquivo*, a partir da Árvore de Diretórios. Diferente do objeto gerado na primeira abordagem, esse objeto possuirá além do atributo *listaIdStorage*, um atributo chamado *codigoHash*. Como resposta, será encaminhado ao cliente além da lista contendo os *Storages* de armazenamento do arquivo, o objeto do tipo *Arquivo* instanciado da Árvore de Diretórios.

De posse da lista que contêm os *Storages* onde o arquivo se encontra armazenado e do código *hash* do arquivo, o próximo passo da Aplicação do Cliente é solicitar esse arquivo aos *Storages*. Diferente da primeira abordagem, onde era disparado uma *Thread* para comunicação com cada *Storage* da lista, nessa abordagem efetuamos uma chamada por vez, i.e., realizamos a comunicação com um *Storage* por vez. Dessa forma, ao receber o arquivo de um *Storage*, armazenamos o arquivo de forma temporário na pasta de armazenamento do cliente e geramos o código *hash* desse arquivo temporário. Caso o código *hash* gerado desse arquivo seja igual ao código *hash* enviado pelo Servidor de Metadados, mantemos o arquivo na pasta do cliente e encerramos a comunicação com os *Storages*. Caso contrário, uma nova chamada é realizada ao próximo *Storage* da lista.

5.4 Forma de Utilização do Sistema

Visando facilitar o uso do sistema para projetos e implementações futuras, criamos uma interface para cada aplicação. Conforme mencionado no Capítulo 4, para a implementação de uma aplicação de servidor baseada na biblioteca BFT-SMaRt, se deve implementar as interfaces *Executable* e *Runnable*. Como solução alternativa adotada no nosso projeto, estendemos a classe abstrata *DefaultSingleRecoverable*, pois a mesma já implementa essas interfaces. Após esses passos, será necessário implementar a interface *InterfaceServidorMetaDados*. Os seguintes métodos pertencentes a *InterfaceServidorMetadados* devem ser implementados, para o uso do nosso projeto:

```
ServiceReplica estabeleceComunicacaoBFT(int idServidor);
ServidorServico criaServidorServico();
```

- estabeleceComunicacaoBFT: método responsável pelo estabelecimento da comunicação com a biblioteca BFT-SMaRt. Recebe como parâmetro, o identificador do servidor;
- criaServidorServico: método responsável pela criação de um novo objeto de serviços para o Servidor. Não recebe nenhum parâmetro.

Após a implementação da interface, será possível utilizar os serviços já criados no nosso sistema, para um Servidor de Metadados. Para isso, se utiliza o objeto do tipo *ServidorServico* já criado. Abaixo segue as assinaturas e descrição de uso dos métodos, que podem ser realizados com o objeto *ServidorServico*:

```
public byte[] criaDiretorio(List<String> diretorioAtual, String nomeNovoDiretorio);
public byte[] removeDiretorio(List<String> diretorioAtual, String nomeDiretorio) ;
public byte[] verificaDiretorio(List<String> diretorioAtual, String nomeDiretorio);
public byte[] listaDados(List<String> diretorioAtual);
public byte[] salvaArquivo(Arquivo novoArquivo, List<String> diretorioAtual, Integer numeroStorages);
public byte[] removeArquivo(Arquivo arquivo, List<String> diretorioAtual);
public byte[] buscaArquivo(List<String> diretorioAtual, String nomeArquivo);
public byte[] buscaListaStorages(Arquivo arquivo);
public byte[] salvaStorage(Storage storage);
```

- criaDiretorio: método para criação de um novo diretório. Recebe como parâmetros, o local para criação do novo diretório e o nome do novo diretório;
- removeDiretorio: método para remoção de um diretório. Recebe como parâmetros, o local de remoção do diretório e o nome do diretório a ser removido;
- verificaDiretorio: método para verificar a existência de um determinado diretório. Recebe como parâmetros, o local de verificação e o nome do diretório a ser verificado;
- listaDados: método para listar os diretórios e arquivos de um determinado local. Recebe como parâmetro, o local para listagem dos dados;
- salvaArquivo: método para salvar um novo arquivo. Recebe como parâmetros, os dados do novo arquivo, o local para armazenamento do novo arquivo e o número de *Storages* para armazenamento do novo arquivo;
- removeArquivo: método para remover um arquivo. Recebe como parâmetros, os dados do arquivo a ser removido e o local para remoção do arquivo;
- buscaArquivo: método para buscar um arquivo. Recebe como parâmetros, o local para busca do arquivo e o nome do arquivo a ser buscado;
- buscaListaStorages: método para buscar os *Storages*, onde está armazenado um determinado arquivo. Recebe como parâmetro, o arquivo a ser verificado;
- salvaStorage: método para adicionar um novo *Storage* na Tabela de *Storages*. Recebe como parâmetro, os dados do novo *Storage*.

Conforme mencionado no Capítulo 4, para a implementação de uma nova aplicação de cliente, que utilize a biblioteca BFT-SMaRt é necessário instanciar um objeto do tipo

ServiceProxy. Para isso, criamos uma interface, que deve ser implementada para o uso do nosso projeto em uma nova aplicação para clientes. Essa interface foi chamada *InterfaceAplicacaoCliente*. Para a implementação da *InterfaceAplicacaoCliente*, os seguintes métodos devem ser implementados:

```
ServiceProxy estabeleceComunicacaoBFT(int idCliente);
Cliente criaCliente(int idCliente, int fNumeroFalhas, String localArmazenamento);
ClienteServico criaClienteServico(ServiceProxy kVProxy, Cliente cliente);
```

- estabeleceComunicacaoBFT: método responsável pelo estabelecimento da comunicação com a biblioteca BFT-SMaRt. Recebe como parâmetro o identificador do cliente;
- criaCliente: método responsável pela criação de um novo objeto do tipo *Cliente*. Recebe como parâmetros, o identificador do cliente, o número de falhas aceitáveis no sistema e o local de armazenamento do usuário;
- criaClienteServico: método responsável pela criação de um novo objeto do tipo *ClienteServico*. Recebe como parâmetros, o objeto de comunicação com a biblioteca BFT-SMaRt e o objeto *Cliente* criado na implementação anterior.

Após a implementação da interface, será possível utilizar os serviços já criados no nosso sistema, para uma nova aplicação para clientes. Para isso, utiliza-se o objeto do tipo *ClienteServico* já criado. Abaixo segue as assinaturas e descrição de uso dos métodos, que podem ser realizados com o objeto *ClienteServico*:

```
public void moveDiretorio(String nomeDiretorio);
public void criaDiretorio(String nomeNovoDiretorio);
public void removeDiretorio(String nomeDiretorio);
public void listaDados();
public void escreveArquivoThread(File arquivoFisico);
public void escreveArquivoHash(File arquivoFisico);
public void removeArquivo(String nomeArquivo);
public void leArquivoThread(String nomeArquivo);
public void leArquivoHash(String nomeArquivo);
```

- moveDiretorio: método para navegação entre diretórios. Recebe como parâmetro, o nome do diretório de destino de navegação;
- criaDiretorio: método para a criação de um novo diretório. Recebe como parâmetro, o nome do novo diretório;
- removeDiretorio: método para remoção de um diretório. Recebe como parâmetro, o nome do diretório;
- listaDados: método para listar os arquivos e diretórios, do atual diretório do cliente. Não recebe nenhum parâmetro;
- escreveArquivoThread: método para a escrita de um novo arquivo, no atual diretório do cliente. Esse método utiliza a primeira abordagem de escrita de arquivos, i.e., não utiliza o conceito de *hash*. Recebe como parâmetro, um objeto do tipo *File* [18], com os dados do arquivo a ser escrito.

- escreveArquivoHash: método para a escrita de um novo arquivo, no atual diretório do cliente. Esse método utiliza a segunda abordagem de escrita de arquivos, i.e., utiliza o conceito de *hash*. Recebe como parâmetro, um objeto do tipo *File*, com os dados do arquivo a ser escrito.
- removeArquivo: método para remoção de um arquivo, do atual diretório do cliente. Recebe como parâmetro, o nome do arquivo a ser removido;
- leArquivoThread: método para a leitura de um arquivo, do atual diretório do cliente. Esse método utiliza a primeira abordagem de leitura de arquivos, i.e., não utiliza o conceito de *hash*. Recebe como parâmetro, o nome do arquivo a ser lido;
- leArquivoHash: método para a leitura de um arquivo, do atual diretório do cliente. Esse método utiliza a segunda abordagem de leitura de arquivos, i.e., utiliza o conceito de *hash*. Recebe como parâmetro, o nome do arquivo a ser lido.

Por último, para utilizar o nosso projeto em novas aplicações para *Storages*, será necessário implementar a *InterfaceStorage*. Para isso os seguintes métodos devem ser implementados:

```
ServiceProxy estabeleceComunicacaoBFT(int idStorage);
Storage criaStorage(int idStorage, int portaConexao, long espacoLivre, String localArmazenamento);
StorageServico criaStorageServico(ServiceProxy KVProxy, Storage storage);
```

- estabeleceComunicacaoBFT: método responsável pelo estabelecimento da comunicação com a biblioteca BFT-SMaRt. Recebe como parâmetro, o identificador do *Storage*;
- criaStorage: método responsável pela criação de um novo objeto do tipo *Storage*. Recebe como parâmetro, o identificador do *Storage*, a porta para conexões via *Socket*, o tamanho máximo de espaço livre no *Storage* e o local para armazenamento dos dados;
- criaStorageServico: método responsável pela criação de um novo objeto do tipo *StorageServico*. Recebe como parâmetro, o objeto de comunicação com a biblioteca BFT-SMaRt e o objeto *Storage* criado na implementação anterior.

Após a implementação da interface, será possível utilizar os serviços já criados no nosso sistema, para uma nova aplicação de *Storages*. Para isso, utiliza-se o objeto do tipo *StorageServico* já criado. Abaixo segue as assinaturas e descrição de uso dos métodos, que podem ser realizados com o objeto *StorageServico*:

```
public boolean iniciaServidor()
private boolean enviaDadosStorageMetaDados(Storage storage);
public boolean aguardaCliente()
public void terminaServidor()
```

- iniciaServidor: método para iniciar um novo *Storage*. Não recebe nenhum parâmetro;
- enviaDadosStorageMetaDados: método para enviar os dados do *Storage* aos Servidores de Metadadados. Recebe como parâmetro, os dados do *Storage*;

- aguardaCliente: método para aguardar e tratar novas requisições de clientes. Não recebe nenhum parâmetro;
- terminaServidor: método para finalizar um *Storage*. Não recebe nenhum parâmetro.

Além dessas classes já citadas, temos a classe *Seguranca*. Essa classe é a responsável pelos métodos de segurança no sistema. Essa classe é a responsável pela implementação do método, para geração de um código *hash* de um arquivo. Para isso, basta instanciar um novo objeto do tipo *Seguranca* e chamar o método *geraHashArquivo*. O método *geraHashArquivo* recebe como parâmetro um *array* de bytes e devolve como resposta, um código *hash* no formato hexadecimal.

```
public static String geraHashArquivo(FileInputStream inputArquivo);
```

5.5 Testes e Resultados

Visando obter e comparar os resultados da utilização das duas abordagens de escrita e leitura, mencionadas no início do capítulo, foram realizados testes no sistema. Definimos como escopo para os nossos testes, capturar os dados de latência e *throughput*, durante a execução das operações de escrita e leitura, de ambas as abordagens e comparar cada resultado encontrado.

Para os testes de latência, as seguintes condições foram utilizadas:

- Para todos os testes, foram utilizados três arquivos com os seguintes tamanhos: 1Kb, 1Mb e 10Mb;
- Apenas um único e mesmo cliente, seria utilizado para realização de todas as operações;
- Para cada operação, foram realizadas 700 requisições, com 50 requisições de *warm up*;
- Foram consideradas todas as medidas dos testes, para coleta dos dados médios e desvio padrão das operações;
- Foi definido para todos os testes, que o sistema suportaria no máximo uma falha;
- A medida utilizada, foi o tempo de execução da operação em microssegundos;
- Todos os testes seriam realizados no melhor caso, i.e., sem falhas;
- Todos os testes foram realizados no mesmo dia e com o mesmo ambiente.

Já para os testes de *throughput*, as seguintes condições foram utilizadas:

- Para todos os testes, foram utilizados três arquivos com os seguintes tamanhos: 1Kb, 1Mb e 10Mb;
- Dez clientes distintos, acessariam a aplicação e realizariam simultaneamente as requisições para cada operação;
- Cada operação foi testada, durante o período de dez minutos;

- Para cada operação, foram realizadas infinitas requisições, durante o período de testes definido;
- Para todos os clientes, foram definidos que o sistema suportaria no máximo uma falha;
- A medida utilizada, foi o número de operações máximas por segundos;
- Todos os testes seriam realizados no melhor caso, i.e., sem falhas;
- Todos os testes foram realizados no mesmo dia e com o mesmo ambiente.

Como ferramenta para auxílio e criação do nosso ambiente de testes, utilizamos o Emulab [30]. A utilização da ferramenta online Emulab é bem simples e de fácil aprendizado. Com alguns passos simples, conseguimos realizar as configurações do ambiente, para realização dos nossos testes. Como a ferramenta é gratuita e compartilhada entre várias instituições de ensino, o seu uso deve ser feito de forma moderada é controlada.

Definimos o nosso ambiente de testes, com as seguintes características:

- No total foram emulados 17 computadores;
  - 10 computadores, para emulação dos clientes;
  - 4 computadores, para emulação dos servidores de metadados;
  - 3 computadores, para emulação dos servidores de armazenamento.
- Todos os computadores possuíam o sistema operacional UBUNTU v.14, 64bits;
- Todos os computadores possuíam 4Gb de Memória RAM e um processador Intel Xeon E5-2630 v3, com 20Mb de Memória Cache e 2.40 GHz;
- Os computadores foram conectados através de um *switch* de 1Gb/s de velocidade.

Latência

Conforme mencionado anteriormente, foram utilizados três arquivos para os testes e realizadas 700 requisições para cada operação, com 50 requisições de *warm up*. As operações testadas foram as seguintes:

- Escrita de um arquivo sem a utilização de *hash* (primeira abordagem);
- Escrita de um arquivo, com a utilização de *hash* (segunda abordagem);
- Leitura de um arquivo, sem a utilização de *hash*;
- Leitura de um arquivo, com a utilização de *hash*.

Primeiramente realizamos os testes de escrita, para ambas as abordagens. Conforme pode ser visto na Figura 5.8, o tempo de escrita de um novo arquivo, utilizando a abordagem sem *hash* é inferior, em relação a abordagem com *hash*.

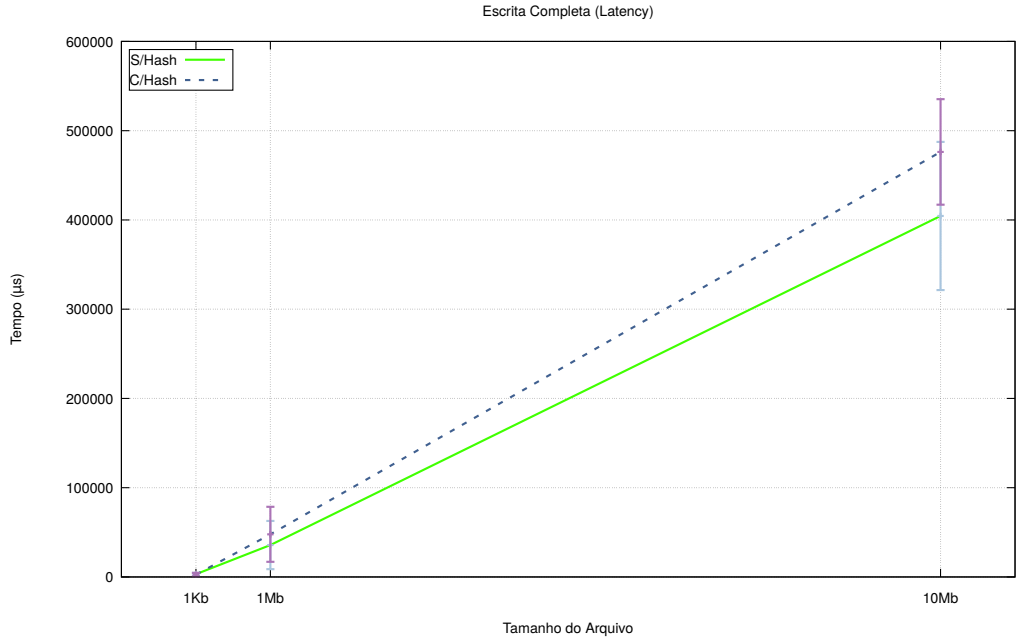

Figura 5.8: Representação dos resultados da latência dos testes de escrita, de forma completa.

Um dos pontos que explica esse resultado é o tempo para geração e processamento dos metadados. Como pode ser visto na Figura 5.9, o tempo para geração dos metadados na primeira abordagem, em todos os casos é bem semelhante. Contudo, na segunda abordagem, o tempo para geração e tratamento dos metadados, varia de acordo com o tamanho do arquivo. Na segunda abordagem, quanto maior o arquivo, maior será o tempo para geração do código *hash* desse arquivo.

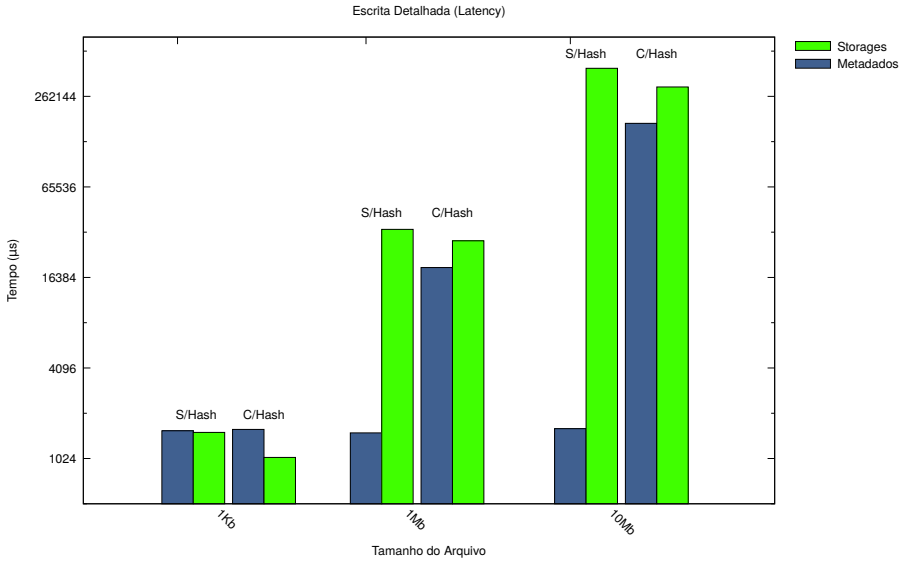

Figura 5.9: Representação dos resultados da latência dos testes de escrita, de forma detalhada.

Um outro ponto importante é o tempo de processamento e envio aos *Storages*. Conforme mencionado no início do capítulo, na primeira abordagem, devemos definir $2f + 1$ *Storages* para o envio do arquivo. Já na segunda abordagem, definimos apenas $f + 1$ *Storages*, para o envio do arquivo. Mesmo utilizando o conceito de *Thread's* e programação paralela, o tempo de envio aos *Storages* na segunda abordagem é menor comparado a primeira abordagem. Um dos motivos que explica esse resultado é a utilização da banda de rede para envio dos dados. Na segunda abordagem, menor é a utilização da banda de rede para envio dos dados. Logo, menor será o tempo, para os *Storages* receberem essa requisição.

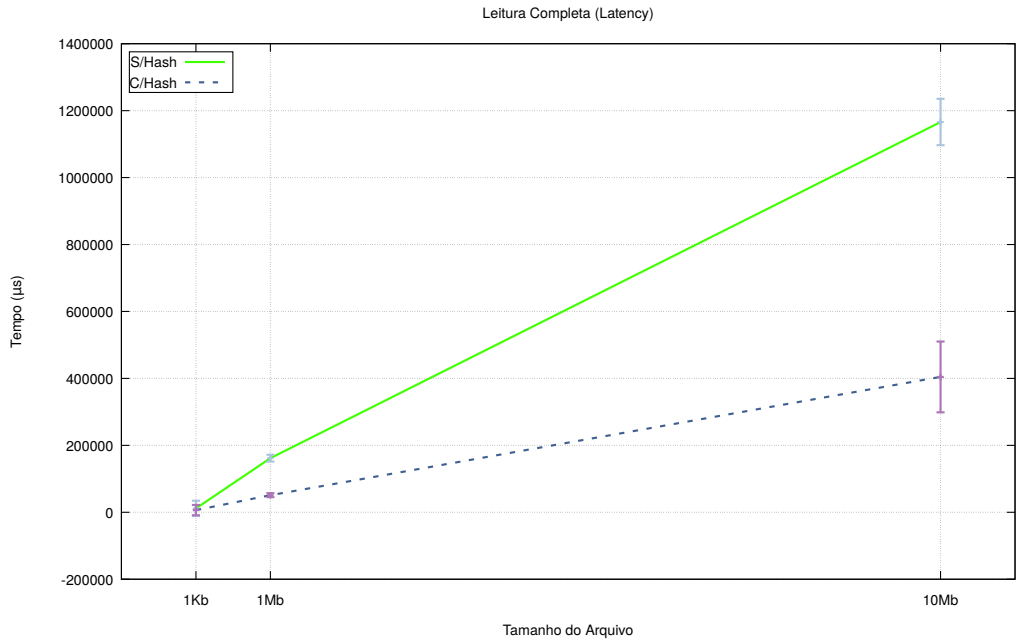

Figura 5.10: Representação dos resultados da latência dos testes de leitura, de forma completa.

Já no caso da leitura, conforme pode ser visto na Figura 5.10, o tempo de leitura de um arquivo, utilizando a abordagem sem *hash* é superior, comparado a abordagem com *hash*.

Conforme mencionado no início do capítulo, uma das condições para os testes, seria a realização dos testes nos melhores casos. No caso da leitura utilizando a segunda abordagem, apenas a comunicação com um único *Storage* foi necessária, i.e., o arquivo recebido do primeiro *Storage* estava íntegro. Já na primeira abordagem, mesmo sendo o melhor caso, conforme descrito durante o capítulo, o cliente recebe uma cópia do arquivo de cada *Storage*, onde o arquivo se encontra armazenado. Dessa forma, o tempo para recebimento e verificação da integridade desse arquivo na primeira abordagem é superior comparado ao tempo da segunda abordagem.

A Figura 5.11 nos mostra os detalhes dos resultados da operação de leitura. Em todos os casos, o tempo de processamento dos metadados é bem semelhante. Contudo, o tempo de processamento nos *Storages* utilizando a abordagem sem *hash* é superior, comparado a abordagem com *hash*.

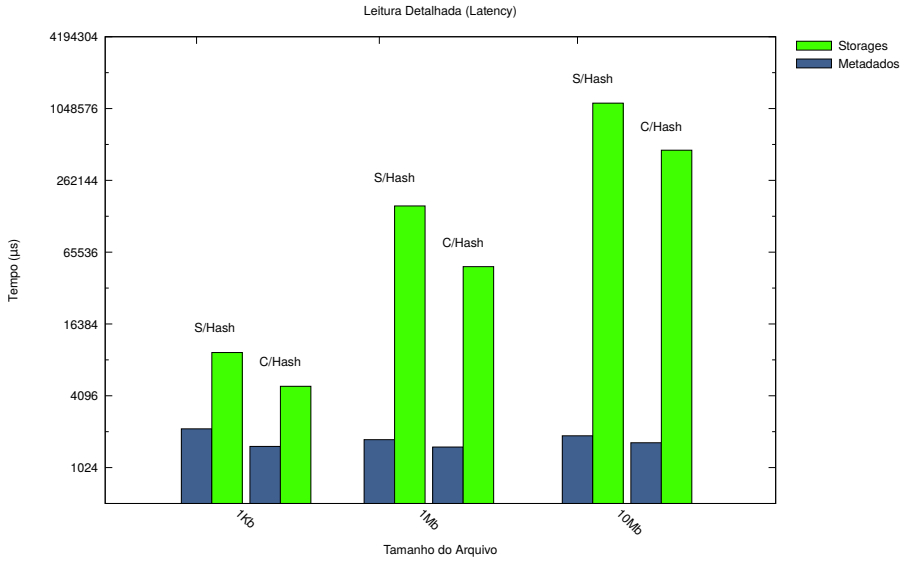

Figura 5.11: Representação dos resultados da latência dos testes de leitura, de forma detalhada.

Throughput

Para os testes de *throughput*, semelhante ao teste de latência, as seguintes operações foram testadas:

- Escrita de um arquivo sem a utilização de *hash* (primeira abordagem);
- Escrita de um arquivo, com a utilização de *hash* (segunda abordagem);
- Leitura de um arquivo, sem a utilização de *hash*;
- Leitura de um arquivo, com a utilização de *hash*.

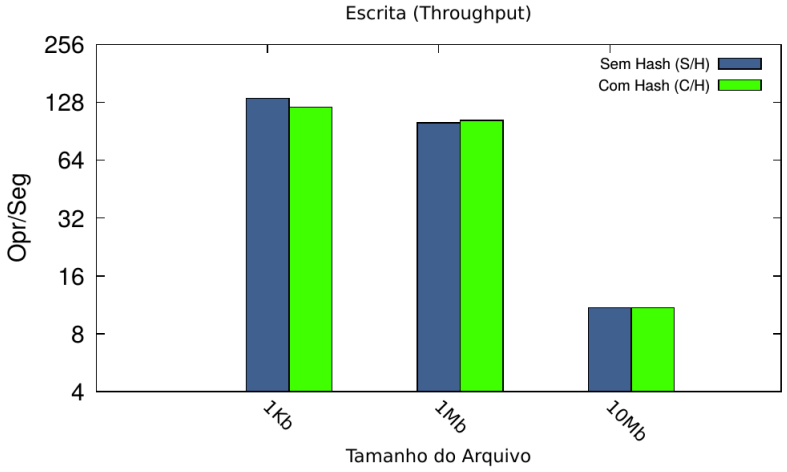

Figura 5.12: Representação dos resultados de *throughput* dos testes de escrita.

Primeiramente realizamos os testes de escrita, para ambas as abordagens. Conforme pode ser visto na Figura 5.12, o número máximo de operações por segundos, varia muito pouco, de uma abordagem para a outra.

Um dos motivos para esse resultado é o fato de termos 10 clientes compartilhando a mesma rede e acessando o sistema simultaneamente. Maior será o tráfego na rede nesses testes, comparados aos testes de latência, onde havia apenas um cliente, acessando o sistema para os testes. Logo, na segunda abordagem, mesmo o tempo para geração do código *hash* e processamento dos metadados, ser relativamente maior comparado à primeira abordagem, o tráfego na rede é menor na segunda abordagem, uma vez que, o número de *Storages* necessários para envio da informação é menor.

Na segunda abordagem, o tempo para processamento do código *hash* é compensado pelo número menor de *Storages* para armazenamento do arquivo. Logo, o número de operações máximas por segundo, será semelhante em ambas as abordagens.

A grande vantagem da utilização da abordagem com *hash*, comparada a abordagem sem *hash* é no momento da leitura. Como pode ser visto na Figura 5.13, em todos os casos, conseguimos realizar em média, o dobro de operações utilizando a abordagem com *hash*, comparada a abordagem sem *hash*.

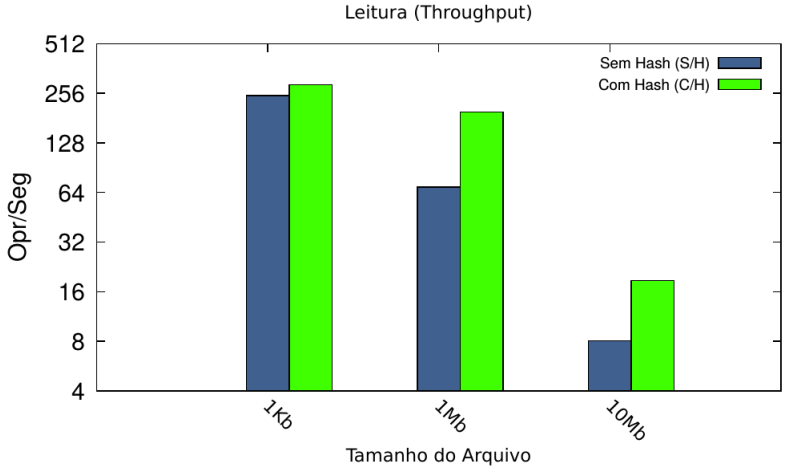

Figura 5.13: Representação dos resultados de *throughput* dos testes de leitura.

Mais uma vez, conforme mencionado anteriormente, estamos usando como condição, os melhores casos para os testes. Na primeira abordagem, mesmo o sistema suportando apenas uma falha, será necessário receber uma cópia do arquivo de todos os *Storages*, uma vez que, o método implementado para verificação da integridade do arquivo na primeira abordagem, depende da comparação entre todos os arquivos armazenados em todos os *Storages*. Já no caso da segunda abordagem, será necessário a comunicação com apenas um único *Storage*, pois o primeiro arquivo recebido já estará íntegro. Contudo, na segunda abordagem será necessário um tempo a mais, para geração do código *hash* do arquivo recebido.

O número de cópias dos arquivos necessários na primeira abordagem, para a verificação da integridade do arquivo, será maior, comparado a segunda abordagem. Com isso, maior será o tempo de processamento e o uso da rede na primeira abordagem, comparada a

segunda. Logo, o número de operações máximas a serem realizadas por segundo, na primeira abordagem, será menor, comparada a segunda abordagem.

Capítulo 6

Conclusão

Em um Sistema de Arquivos Distribuído (SAD) é importante que o sistema seja seguro e confiável. Além disso, o sistema deve ser escalável, ou seja, deve haver a possibilidade de adicionar, de modo simples, novos *hardwares* na rede, para compor o sistema. A redundância nesses sistemas também é um ponto importante, pois o fato de uma máquina do sistema ficar indisponível por qualquer tipo de problema, não deve inviabilizar o funcionamento do sistema como um todo.

O uso de um parque de computadores apenas para o tratamento dos metadados e o uso de um outro parque de computadores para o armazenamento do arquivo, facilita o tráfego de informações e a velocidade do sistema. Apesar disso, uma maior atenção deve ser feita em relação a segurança no sistema, visto que com dois grandes parques de computadores, maior será o controle e gerenciamento desse sistema.

A proposta do sistema apresentado no projeto, possuí como características o uso de uma biblioteca tolerante a falhas bizantinas e falhas por *crash*. Além disso, possuí um sistema altamente escalável, com a fácil integração de novos componentes ao sistema. Abordamos alguns dos métodos que podem ser utilizados para a escrita e leitura dos arquivos no sistema. Dentre esses métodos, dois foram utilizados e comparados no sistema. Um utilizando *funções hash* e outro não. Entre os resultados obtidos, destaca-se o uso da técnica com *hash*, ser superior nos melhores casos, no momento da leitura de um arquivo.

A implementação do sistema, vai ao encontro com o objetivo geral da pesquisa. Como forma de sustentar e atender esse objetivo, revisamos os objetivos específicos, que foram alcançados da seguinte forma:

- Os conceitos de segurança de funcionamento de um Sistema de Arquivos Distribuído, foram descritos no Capítulo 2 e Capítulo 3, respectivamente;
- Os conceitos da biblioteca BFT-SMaRt, foram descritos no Capítulo 4;
- O desenvolvimento de uma hierarquia lógica de diretórios do sistema, foi descrito no Capítulo 5;
- O desenvolvimento das operações do sistema para manipulação dos arquivos e diretórios, foi descrito no Capítulo 5;
- Os testes e a análise dos resultados, foram feitos no Capítulo 5;

Todo o código fonte e os arquivos de configurações do sistema, estão disponíveis no seguinte repositório:

- https://github.com/guilherme110/projetoFinal.git

6.0.1 Trabalhos Futuros

Novas implementações podem ser feitas no sistema, a fim de melhorar o seu desempenho e sua segurança. Abaixo segue uma breve descrição de cada melhoria que pode ser realizada no sistema:

- Implementar o controle de acesso dos clientes ao sistema. Hoje, caso um usuário salve um arquivo no sistema, todos os outros clientes que tenham acesso à aplicação, podem ter acesso a esse arquivo. Seria necessário implementar um mecanismo para o usuário criador do arquivo, gerenciar o controle de acessos ao arquivo ou diretório;
- Melhorias no algoritmo de definição de melhores *Storages*, para o armazenamento do arquivo. Para o *Storage* ser definido como o melhor *Storage*, o *Storage* precisa apenas ter o tamanho de espaço maior ou igual ao tamanho do arquivo. Uma possível melhoria, seria a identificação de qual *Storage* está mais próximo fisicamente do cliente. Com isso, o tempo para envio dos dados ao *Storage*, seria menor;
- Implementar um mecanismos de *rollback* no sistema. Hoje, caso o cliente realize uma requisição de escrita, de um novo arquivo no sistema, primeiramente a Árvore de Diretórios e a Tabela de *Storages* são atualizadas com essa nova requisição. Caso algum problema ocorra na etapa de envio dos dados aos *Storages*, nenhuma mensagem de problema é enviada aos Servidores de Metadados. Logo, as informações de estado da aplicação ficam desatualizadas. Seria necessário implementar um mecanismo de bloqueio do arquivo nos Servidores de Metadados e o *Storage* emitir uma mensagem de confirmação de salvamento dos arquivos aos Servidores de Metadados. Somente após o recebimento da mensagem de confirmação, o arquivo seria desbloqueado para novas consultas;
- Implementar uma interface visual para navegação e operação no sistema. Atualmente, toda navegação no sistema e feita através de comandos no terminal.

Referências

[1] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. Proceedings of the 5th symposium on operating systems design and implementation farsite: Federated, available, and reliable storage for an incompletely trusted environment. (1):925–1023, June 2012. 16, 27

[2] Apache Commons IO Apache. Apache common user guide. *https://commons.apache.org/proper/commons-io/description.html* *Last Accessed: June 5th*, 2016. 42

[3] Algirdas Avižienis. Design of fault-tolerant computers. In *Proceedings of the November 14-16, 1967, fall joint computer conference*, pages 733–743. ACM, 1967. 14

[4] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004. vii, 9, 14, 22

[5] Maurice J Bach. *The design of the UNIX operating system*, volume 5. Prentice-Hall Englewood Cliffs, NJ, 1986. 13, 35, 39

[6] Leandro M Barros. Suporte a programaçao genérica em linguagens. 26

[7] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 355–362. IEEE, 2014. 23, 24, 25, 26, 30, 33, 35, 36, 38

[8] Alysson Neves Bessani. Bft-smart: Uma ferramenta robusta para replicação máquina de estados. (1):1011–1018, August 2014. 23

[9] BFT-SMaRt. High-performance byzantine fault-tolerant state machine replication , 1999. 24

[10] R. E. Blahut. Theory and practice of error control codes. 1983. 16

[11] Bruno Filipe Cabo Verde de Brito et al. *Evolução da biblioteca para replicação tolerante a faltas bizantinas BFT-SMaRt*. PhD thesis, 2011. 24

[12] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999. 16

[13] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Sistemas Distribuídos-: Conceitos e Projeto*. Bookman Editora, 2013. 5, 7, 12, 27

[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003. 28

[15] Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java concurrency in practice*. Addison-Wesley, 2006. 41

[16] Douglas Kramer. Api documentation from source code comments: a case study of javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153. ACM, 1999. 32

[17] James F Kurose, Keith W Ross, Arlete Simille Marques, and Wagner Luiz Zucchi. *Redes de Computadores ea Internet: uma abordagem top-down*. Pearson, 2010. 24, 29, 39

[18] Java SE Oracle. The java docs. *https://docs.oracle.com/javase/7/docs/api/java/io/File.html* *Last Accessed: June 10th*, 2016. 46

[19] Java SE Oracle. The java tutorials. *https://docs.oracle.com/javase/tutorial/java/concepts/interface.html* *Last Accessed: June 10th*, 2016. 24, 32

[20] Java SE Oracle. The java tutorials. *https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html* *Last Accessed: June 10th*, 2016. 25, 38

[21] Java SE Oracle. The java tutorials. *https://docs.oracle.com/javase/tutorial/java/concepts* *Last Accessed: June 10th*, 2016. 32, 42

[22] Java SE Oracle. The java tutorials. *https://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html* *Last Accessed: June 10th*, 2016. 39

[23] Nicolas Sklavos and O Koufopavlou. On the hardware implementations of the sha-2 (256, 384, 512) hash functions. In *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*, volume 5, pages V–153. IEEE, 2003. 31, 42

[24] Ian Sommerville and Pete Sawyer. *Requirements engineering: a good practice guide*. John Wiley & Sons, Inc., 1997. 32, 36

[25] Andrew S Tanenbaum. *Sistemas operacionais modernos*. Pearson Prentice Hall, 2003. 23, 26, 28

[26] Andrew S Tanenbaum and Maarten Van Steen. Distributed systems. 2007. 5, 7

[27] Andrew S Tanenbaum and Albert S Woodhull. *Sistemas Operacionais: Projetos e Implementação*. Bookman, 2006. 37

[28] Simão Sirineo Toscani and Alexandre da Silva Carissimi. *Sistemas operacionais e programação concorrente*. Sagra Luzzatto, 2003. 8, 9

[29] Ludwig Von Bertalanffy. *Teoria geral dos sistemas*, volume 351. Vozes, 1975. 23

[30] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of 5th Symp. on Operating Systems Design and Implementations*, 2002. 33, 49

[31] Steve Wilbur. Distributed systems security. 2(6), 2000. 18